

ПРОГРАММИРОВАНИЕ В СРЕДЕ

Л.М. РОМАНОВСКАЯ
Т.В. РУСС
С.Г. СВЯТКОВСКИЙ

СИ ДЛЯ

ПЭВМ ЕС



Л.М. РОМАНОВСКАЯ
Т.В. РУСС
С.Г. СВИТКОВСКИЙ

**ПРОГРАММИРОВАНИЕ
В СРЕДЕ Си
для
ПЭВМ
ЕС**



МОСКВА
"ФИНАНСЫ И СТАТИСТИКА"
1992

ББК 32.973-01

Р69

УДК 681.3.06

Рецензент канд. техн. наук
С.А.Котов

Романовская Л.М. и др.

Р69 Программирование в среде Си для ПЭВМ ЕС/Л.М.Романовская, Т.В.Русс, С.Г.Свитковский. – М.: Финансы и статистика, 1992. – 352 с.: ил.

ISBN 5-279-00506-1.

Рассматривается конкретная реализация языка Си для персональных ЭВМ Единой системы в операционной системе Альфа-ДОС. Приводится описание применения функций библиотеки Си при программировании ввода-вывода, графических приложений и работы с окнами, а также возможностей интегрированной среды разработки Си-программ.

Для профессиональных программистов и специалистов по вычислительной технике, а также студентов вузов соответствующих специальностей.

Р 2404090000—001 131-91
010(01)-92

ББК 32.973-01

ISBN 5-279-00506-1

© Л.М.Романовская, Т.В.Русс,
С.Г.Свитковский, 1991

ВВЕДЕНИЕ В СИСТЕМУ ПРОГРАММИРОВАНИЯ СИ

О языке Си

Появление персональных ЭВМ значительно расширило круг пользователей вычислительной техники, который включает теперь не только профессиональных программистов, но и специалистов различных областей деятельности. Изменился и состав программного обеспечения ЭВМ, в котором большое место занимают готовые прикладные системы, облегчающие пользователю применение ЭВМ для различных видов работ. Тем не менее нередко возникает потребность расширить или модифицировать некоторые функции готовой системы. Саму прикладную систему тоже необходимо программировать на каком-то языке. Таким языком программирования для персональных ЭВМ стал язык Си — язык программирования системных и прикладных программ для профессиональных программистов. Конечно, языком Си могут пользоваться не только профессионалы высокого класса. Язык Си не сложнее Паскаля или Фортрана. Но вместе с тем он обладает свойствами, которые привлекательны прежде всего для профессионалов: это лаконичность языка, сочетающаяся с большой мощностью языка, это возможность использовать машинно-ориентированные средства, с одной стороны, и создавать мобильные программы, с другой стороны. Язык включает все управляющие конструкции структурного программирования, что способствует созданию программ в хорошем стиле. Однако из-за того, что язык разрешает практически все, его нельзя считать языком надежного программирования, и вся ответственность за надежность создаваемой на языке Си программы лежит на программисте. Поэтому программист должен хорошо знать особенности языка и реализации, чтобы со знанием дела их использовать.

Имеется уже довольно большое число публикаций по языку Си на русском языке [1-6]. Однако большей частью это или учебники, не дающие полного описания языка, или краткие справочники. Все они связаны с реализацией языка в операционной системе UNIX.

Хотя язык позволяет создавать мобильные программы, при практическом переносе Си-программы из одной вычислительной среды в другую могут потребоваться изменения в программе, связанные с неодинаковыми именами определенных для конкретной реализации функций, констант, структур данных, файлов. Однако правильная организация программы и наличие в любом компиляторе языка Си препроцессора облегчают проблему переноса.

Данная книга содержит относительно полное описание языка Си, реализованного в системе программирования Си на персональных ЭВМ Единой системы для операционной системы Альфа-ДОС и совместимых с ними систем. Необходимую информацию о персональных ЭВМ Единой системы и работе в операционной системе можно найти в [7,8].

В качестве введения в программирование в среде Си для ПЭВМ ЕС рассмотрим пример простой программы, которую можно ввести и выполнить.

```
/* Пример программы копирования исходной строки в
   результирующую и ее печать */
#define КОНЕЦ '\0'          /* Определение константы */
#include <stdio.h>           /* Включение файла, содержащего
                               прототип стандартной функции printf */
main( )                     /* Определение функции main */
{
    char рез_строка[20];     /* Объявление массива символов */
    char исх_строка[] = "Пример программы";
                               /* Объявление массива символов с
                               начальным значением */
    int i=0;                 /* Объявление переменной целого
                               типа, которая будет индексом массива */
    while ((рез_строка[i] = исх_строка[i]) != КОНЕЦ)
        i++;                 /* Операторы программы */
    printf ("%s", рез_строка);
}
```

Первые две строки программы содержат директивы препроцессора. Первая директива `#define` определяет имя `КОНЕЦ` для нулевого символа конца строки `'\0'`. Препроцессор заменит все появления константы `КОНЕЦ` в программе символьной константой `'\0'`. Вторая директива `#include` предписывает препроцессору включить содержимое файла с именем `stdio.h` в указанное место исходной Си-программы. Этот файл поставляется в дистрибутивной системе программирования Си и содержит определения констант, типов данных, прототипы функций, связанных с вводом-выводом (в частности, для этой программы он позволяет использовать функцию `printf` для вывода информации на дисплей). Си-программа всегда должна содержать функцию с зарезервированным именем `main`, с которой начинается выполнение программы. Третья строка программы содержит заголовок функции `main( )` без параметров. Далее следуют объявления переменных и операторы,

выполняющие копирование содержимого исходной строки в результирующую и вывод результирующей строки на дисплей.

Каждое объявление и каждый оператор в Си-программе заканчивается точкой с запятой. Точка с запятой не ставится в конце группы операторов, заключенных в фигурные скобки, определения функции и директивы препроцессора. В данном примере использованы русские идентификаторы, но ключевые слова языка (`char`, `int`, `while`) применяются в английской нотации. Мы не предлагаем для них русской нотации по нескольким причинам. Во-первых, ключевых слов в языке Си всего лишь 32. Их русские эквиваленты не стандартизованы, длиннее при написании без сокращений и менее mnemonic при сокращении. Во-вторых, каждый пользователь может ввести свои собственные эквиваленты ключевых слов, используя возможности препроцессора.

Состав системы программирования

Создание текста исходной программы, компиляция, редактирование, отладка и пуск программы могут быть выполнены в системе программирования Си (СП Си) с использованием средств интегрированной среды (причем все этапы подготовки и выполнения программы можно осуществить, не выходя из интегрированной среды) или с использованием автономных компонентов операционной системы и системы программирования Си. Упрощенная схема взаимодействия компонентов приведена на рис. 1.

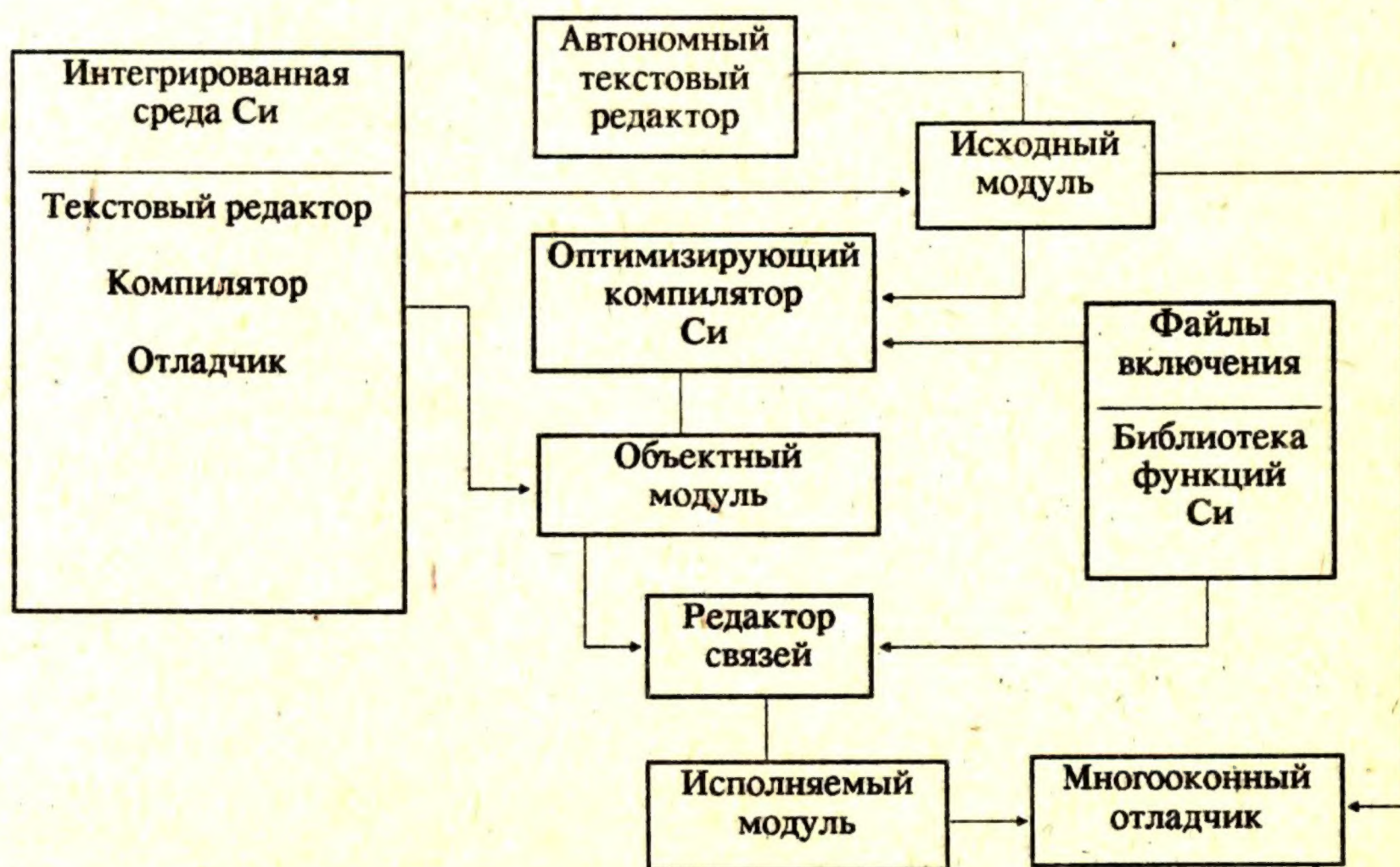


Рис. 1. Упрощенная схема взаимодействия компонентов системы программирования

Интегрированная среда

Пользователь управляет работой в интегрированной среде с помощью предлагаемых системой меню с набором команд. Выбирая ту или иную команду из меню, пользователь может не знать, с каким компонентом он собирается работать, — он выбирает работу, которую нужно выполнить.

Назначение интегрированной среды — обеспечить пользователю удобную возможность получить как можно быстрее работающую Си-программу.

После загрузки в память интегрированная среда по умолчанию устанавливается на режим редактирования текста, т.е. программист может сразу начинать ввод текста исходной программы. Затем набранный текст может быть откомпилирован и выполнен. Если при компиляции обнаруживаются ошибки в исходной программе, среда снова устанавливается на редактирование текста, и можно скорректировать строки, содержащие ошибки.

Текст исходной Си-программы, создаваемый при работе в интегрированной среде, может быть сохранен для последующего использования или в интегрированной среде, или с автономным оптимизирующим компилятором системы программирования Си. С другой стороны, текст исходной Си-программы может быть создан вне интегрированной среды с помощью любого другого подходящего текстового редактора и использоваться при работе в среде или с оптимизирующим компилятором. Более того, интегрированная среда может быть настроена на использование другого текстового редактора взамен встроенного в среду.

Для отладки Си-программ в интегрированной среде имеется встроенный отладчик, с помощью которого можно устанавливать точки прерывания, следить за значениями выражений и переменных, трассировать программу, включать пошаговое выполнение.

При желании история отладки программы (команды отладки и вводимая пользователем входная информация проведенного сеанса отладки) может быть сохранена во внешней памяти и затем воспроизведена при необходимости в этом же или другом сеансе работы с изменением или без изменения входной информации.

С компилятором интегрированной среды можно работать в двух режимах: в режиме оконных меню и режиме командной строки.

Работая в режиме оконных меню, пользователь может выбирать нужное меню и команды меню, используя клавиши управления перемещением курсора или манипулятор графической информации КОЛОБОК. При работе в режиме командной строки команда вызова компилятора набирается на клавиатуре дисплея полностью со всеми операндами, определяющими требуемые режимы работы компилятора, и интерактивная связь пользователя с компилятором отсутствует.

Однако при работе в режиме командной строки пользователю доступны все возможности, предоставляемые компилятором, без ограничений, которые имеются при работе в интегрированной среде.

Интегрированная среда предоставляет пользователю мощную контекстно чувствительную систему получения справочной информации о всех аспектах работы в среде: назначении команд в меню, элементах языка Си, формате вызова и назначении функций библиотеки, возможных причинах сообщений об ошибках, выдаваемых компилятором и редактором связей. Для получения справки достаточно подвести курсор к интересующей теме и нажать функциональную клавишу F1.

Для облегчения ведения многомодульного программного проекта в интегрированной среде заложена возможность отслеживать соответствие времени последнего изменения исходного и объектного модуля из программного списка, представляющего разрабатываемый проект. При возникновении несоответствия может быть произведено автоматическое пересоздание исполняемой программы. При этом средой создается файл во внешней памяти, который имеет формат, пригодный для обработки утилитой MAKE. Благодаря этому сопровождение многомодульного проекта может выполняться и вне интегрированной среды.

Оптимизирующий компилятор

При разработке промышленных программных продуктов целесообразно сочетать работу в интегрированной среде с использованием оптимизирующего компилятора Си: на начальных стадиях разработки программ удобно использовать интегрированную среду для быстрого получения результата, а на завершающей стадии применять оптимизирующий компилятор для получения эффективной исполняемой программы. Оптимизирующий компилятор позволяет получить более эффективный объектный код по длине кода или времени выполнения в соответствии с указанными режимами оптимизации. Некоторые виды оптимизации выполняются всегда независимо от указанных режимов работы компилятора.

Оптимизирующий компилятор позволяет также запросить использование аппаратных возможностей процессора (например, генерацию команд процессора 286 или арифметического сопроцессора). В отличие от компилятора интегрированной среды оптимизирующий компилятор может создать файл объектного модуля в ассемблерном виде, который пригоден для обработки макроассемблером MASM, что дает пользователю возможность улучшить или изменить сгенерированный компилятором код на уровне ассемблерного кода. Система обеспечивает также возможность связи Си-программ с программами на ассемблере, Фортране, Паскале, которые объединяются с Си-программами на этапе редактирования связей.

Библиотека функций

Библиотека функций Си является важным компонентом системы программирования в силу того, что язык Си не имеет встроенных операторов для управления вводом-выводом, процессами, памятью и т.п. Все эти задачи обеспечиваются набором функций библиотеки Си, которая поставляется в составе системы программирования. Пользователь может воспользоваться предоставляемыми функциями или разработать собственные функции для этих же или других задач. В составе библиотеки имеются логические разделы, содержащие совокупности функций для выполнения таких задач, как управление каталогом, управление файлами, ввод и вывод, управление процессами, распределение памяти, поиск и сортировка, операции над строками, операции над буферами, проверка и преобразование символов, преобразование данных, получение времени, математические функции и вызовы операционной системы. Отдельными физическими частями библиотеки, которые могут включаться в объединенную библиотеку во время генерации системы программирования, являются библиотеки для операций с плавающей точкой, библиотеки графических функций, диаграмм и шрифтов и библиотека функций для работы с окнами. В зависимости от выбранных для работы моделей памяти (малой, средней, компактной, большой или сверхбольшой) создаются соответствующие варианты объединенной библиотеки.

Логически к библиотеке функций относятся также файлы включения (имена этих файлов имеют расширение .h), которые содержат в виде фрагментов на языке Си определения символических констант, стандартных типов, глобальных переменных, используемых функциями, а также объявления стандартных функций библиотеки Си. Кроме того, некоторые функции библиотеки Си реализованы как макроопределения, т.е. они заменяются макрорасширениями при обработке препроцессором во время трансляции Си-программы, и поэтому тоже находятся в файлах включения.

Многооконный отладчик

Автономный многооконный отладчик используется для отладки Си-программ, обработанных компилятором интегрированной среды или оптимизирующим компилятором с установленным режимом отладки. Многооконный отладчик обеспечивает возможность отладки на уровне исходной Си-программы и на уровне ассемблерного кода и имеет два режима работы: режим оконных меню и команд, при котором может быть открыто несколько окон для наблюдения и управления различного рода информацией отладки, и режим диалоговых ко-

манд, вводимых с клавиатуры дисплея и позволяющих задавать в командах операнды.

Многооконный отладчик, кроме возможностей, реализованных во встроенном отладчике интегрированной среды Си, обеспечивает также возможность устанавливать условные точки прерывания, точки прерывания по изменению значений переменных, возможность модифицировать память, следить за значением и модифицировать содержимое регистров и флажков.

Набор команд многооконного отладчика совместим с командами системного отладчика SYMDEV.

Организация книги

Книга состоит из четырех частей. В первой части (главы 1-7) дается описание языка Си, реализованного в системе программирования Си. Это описание не является строго формализованным даже в части синтаксиса языка. Мы здесь следуем сложившейся традиции в публикациях, касающихся языка Си. Тем не менее поясним систему обозначений, принятую при описании синтаксиса языка.

Выделенные жирным шрифтом ключевые слова и специальные символы должны в Си-программе записываться так, как они показаны в формате синтаксической единицы.

Курсивом в форматах представлены слова, обозначающие синтаксические понятия, определяемые где-либо в тексте.

Квадратные скобки, не выделенные жирным шрифтом, обозначают, что заключенный в них элемент может быть опущен при использовании в Си-программе. Альтернативные варианты формата представлены с новой строки.

Круглые, квадратные и фигурные скобки являются элементами языка и должны использоваться так, как показано в форматах. Круглые скобки заключают списки параметров и аргументов функции, а также используются для группирования выражений в целях указания порядка их вычисления.

Квадратные скобки используются в определении массивов и элементов массивов.

Фигурные скобки служат для образования составных операторов, ограничивают тело функции, а также список элементов структуры, смеси или перечисления при их определении.

За элементом формата, заключенным в квадратные скобки, может следовать многоточие, которое означает, что этот элемент может быть неоднократно повторен. В этом случае многоточие является элементом метасинтаксиса.

Список параметров функции в объявлении или определении функции может заканчиваться запятой и многоточием (,...) и в этом случае

многоточие является элементом синтаксиса языка и выделено жирным шрифтом. В примерах программ многоточие может использоваться для обозначения опущенной части примера, которая не является существенной в данном случае.

Вторая часть книги содержит некоторые рекомендации по программированию, отражающие специфические особенности реализации языка Си на 16-разрядной ПЭВМ с сегментной архитектурой. Здесь же описывается организация связи Си-программы с программами на других языках (ассемблер, Фортран, Паскаль).

Третья часть книги показывает применение функций библиотеки Си при программировании ввода-вывода, графических программ и программ работы с окнами. Эта часть содержит ряд практически полезных программ, которыми может воспользоваться читатель при разработке собственных программ.

В четвертой части описываются возможности и техника работы с меню в интегрированной среде Си при создании, корректировке, компиляции, отладке и выполнении Си-программы. В этой части, так же как и в остальных частях книги, используются коды и названия клавиш клавиатуры ПЭВМ ЕС1842. Названия клавиш обозначаются прописными буквами; комбинации клавиш соединяются знаком плюс, например УПР+Z. Для других типов ПЭВМ названия клавиш и некоторые коды могут отличаться от используемых здесь.

В приложении 1 можно найти полный перечень кодов и названий клавиш клавиатуры ПЭВМ ЕС1842 и эквивалентные им названия на клавиатуре ПЭВМ ЕС1841 и совместимых с ПЭВМ ЕС зарубежных ПЭВМ, если они отличаются.

Приложение 2 содержит краткий справочник функций библиотеки Си.

Часть 1

ЯЗЫК ПРОГРАММИРОВАНИЯ СИ

Глава 1

ЭЛЕМЕНТЫ ЯЗЫКА

1.1. КЛАССИФИКАЦИЯ СИМВОЛОВ ЯЗЫКА

В зависимости от назначения и использования множество символов языка Си подразделяется на несколько групп.

1. Символы, используемые для образования ключевых слов и идентификаторов. Эту группу символов составляют прописные и строчные буквы английского и русского алфавитов, арабские цифры и символ подчеркивания (табл. 1.1).

Таблица 1.1

Символы для образования ключевых слов и идентификаторов

Прописные буквы латинского алфавита	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Строчные буквы латинского алфавита	a b c d e f g h i j k l m n o p q r s t u v w x y z
Прописные буквы русского алфавита	А Б В Г Д Е Ё Ж З И Й К Л М Н О П Р С Т У Ф Х Ц Ч Ш Щ Ъ Ы Ь Э Ю Я
Строчные буквы русского алфавита	а б в г д е ё ж з и й к л м н о п р с т у ф х ц ч ш щ ъ ы ь э ю я
Арабские цифры	0 1 2 3 4 5 6 7 8 9
Символ "подчеркивание"	—

Одинаковые прописные и строчные буквы считаются различными символами.

2. Знаки пунктуации и специальные символы (табл. 1.2).

Эти символы используются для различных целей — от организации текста программы до определения действий, которые будут выполнены компилятором или откомпилированной программой.

3 Пробельные символы. К этой группе символов относятся пробел,

символы табуляции, перевода строки, возврата каретки, перевода формата и новой строки. Эти символы отделяют объекты, определенные пользователем, такие как константы и идентификаторы, от других объектов программы. Последовательность пробельных символов рассматривается компилятором как один пробельный символ.

Таблица 1.2

Знаки пунктуации и специальные символы

Символ	Наименование	Символ	Наименование
,	Запятая	)	Круглая правая скобка
.	Точка	{	Фигурная левая скобка
;	Точка с запятой	}	Фигурная правая скобка
:	Двоеточие	<	Меньше
?	Вопросительный знак	>	Больше
'	Апостроф	[	Квадратная левая скобка
!	Восклицательный знак	]	Квадратная правая скобка
	Вертикальная черта	#	Номер
/	Дробная черта	%	Процент
\	Обратная дробная черта	&	Коммерческое И (амперсанд)
~	Черта сверху (тильда)	↑	Логическое НЕ (стрелка вверх)
*	Звездочка	-	Минус
(	Круглая левая скобка	=	Равно
+	Плюс	"	Кавычки

Комбинация клавиш УПР+Z создает код, который рассматривается как признак конца файла. Компилятор игнорирует любой текст, следующий за символом УПР+Z.

Кроме этих групп символов, в языке Си используются управляющие последовательности. Управляющие последовательности — это специальные символьные комбинации, позволяющие представить пробельные и неграфические символы в символьных и строковых константах. Управляющая последовательность состоит из обратной дробной черты, за которой следует буква или комбинация цифр (табл. 1.3).

Последовательности \ddd и \xddd позволяют представлять любой

символ из набора кодов ПЭВМ как последовательность восьмеричных или шестнадцатеричных цифр соответственно. Например, символ возврата каретки может быть задан различными способами:

`\r` — обычная управляющая последовательность;

`\015` — восьмеричная управляющая последовательность;

`\x00D` — шестнадцатеричная управляющая последовательность.

Таблица 1.3

Управляющие последовательности

Управляющая последовательность	Наименование	Шестнадцатеричное значение
<code>\a</code>	Звонок	7
<code>\b</code>	Возврат на шаг	8
<code>\t</code>	Горизонтальная табуляция	9
<code>\n</code>	Новая строка	A
<code>\v</code>	Вертикальная табуляция	B
<code>\r</code>	Возврат каретки	D
<code>\f</code>	Перевод формата	C
<code>\"</code>	Кавычки	22
<code>\'</code>	Апостроф	27
<code>\\</code>	Обратная дробная черта	5C
<code>\ddd</code>	Символ из набора кодов ПЭВМ в восьмеричном представлении	
<code>\xdd</code>	Символ из набора кодов ПЭВМ в шестнадцатеричном представлении	ddd

В символьном литерале нет необходимости задавать все три цифры управляющей последовательности. Символ возврата каретки в символьной константе можно также задать как `\15`, `\xD`. В строковых константах же целесообразнее задавать все три цифры управляющей последовательности. В противном случае, если символ, следующий за управляющей последовательностью, случайно окажется восьмеричной или шестнадцатеричной цифрой, компилятор воспримет этот символ как часть управляющей последовательности. Например, чтобы напечатать два слова "Good afternoon", разделенные восьмью пробелами, необходимо записать строку в виде "Good\x009afternoon". Если использовать строку "Good\x09afternoon", то на печати появится

"Good^дfternoon", так как компилятор воспринимает последовательность \x09a как символ "д".

Если обратная дробная черта предшествует символу, не включенному в табл. 1.3 и не являющемуся цифрой, то эта черта игнорируется, а символ представляется как литеральный. Например, символ \h представляет символ h в строковой или символьной константе.

Кроме определения управляющих последовательностей, символ обратной дробной черты (\) используется также как символ продолжения. Если за обратной дробной чертой следует символ новой строки, то оба символа игнорируются, а следующая строка рассматривается как часть предыдущей. Это свойство используется в директиве препроцессора #define для записи определений, состоящих более чем из одной строки. Оно может быть использовано также для создания длинных строковых литералов.

1.2. ЛЕКСЕМЫ

Лексема — это единица текста программы, которая имеет определенный смысл для компилятора и которая не может быть разбита в дальнейшем. В языке Си лексемами являются:

- знаки пунктуации, такие, как квадратные ([]), фигурные ({ }), угловые (< >), круглые скобки, запятая (,), точка с запятой (;);
- знаки операций;
- константы;
- идентификаторы;
- ключевые слова.

Границы лексем определяются пробельными символами и другими лексемами, такими, как знаки пунктуации и знаки операций. Если лексем не отделены друг от друга пробельными символами, компилятор, выделяя отдельную лексему, объединяет столько символов, сколько возможно, прежде чем перейти к следующей лексеме. В этом случае лексем могут быть проинтерпретированы неверно.

Например, разбирая выражение x---y, компилятор вначале выделяет из трех знаков "--" самый длинный из возможных знаков операций (--), а затем обрабатывает оставшийся знак минус как знак операции вычитания. Выражение проинтерпретируется как (x--)-(y), а не (x)-(--y). Чтобы избежать неоднозначности и обеспечить правильное вычисление выражений, рекомендуется использовать пробельные символы и круглые скобки для разделения лексем.

Знаки операций

Знак операции — это один или более специальных символов, определяющих действие, которое должно быть выполнено над величинами.

В многосимвольных знаках операций не должно быть пробельных символов. Знаки операций делятся на унарные, бинарные и тернарные по количеству операндов, участвующих в операции.

Знаки операций приведены в табл. 3.1-3.3 в главе 3, где содержится описание всех операций языка.

Константы

Константами называются неизменяемые величины в программе. В языке Си различают четыре типа констант: целые константы, константы с плавающей точкой, символьные константы и строковые литералы.

Целая константа — это десятичное, восьмеричное или шестнадцатеричное число, которое представляет целую величину в одной из следующих форм: десятичной, восьмеричной или шестнадцатеричной.

Десятичная константа имеет формат:

цифры

где *цифры* представляют одну или несколько десятичных цифр от 0 до 9, первая из которых не должна быть 0.

Восьмеричная константа имеет формат:

0цифры

где *цифры* представляют одну или несколько восьмеричных цифр (от 0 до 7). Ведущий нуль обязателен.

Шестнадцатеричная константа имеет формат:

0хцифры

0Хцифры

где *цифры* представляют одну или несколько шестнадцатеричных цифр (цифры от 0 до 9 и строчные или прописные латинские буквы от "a" до "f"). Все шестнадцатеричные константы должны начинаться с символов 0х или 0Х.

Примеры целых констант приведены в табл. 1.4.

Таблица 1.4

Примеры целых констант

Десятичная константа	Восьмеричная константа	Шестнадцатеричная константа
16	020	0x10
127	0177	0x7f

Целые константы всегда представляют положительные величины. Если требуется отрицательная величина, то надо сформировать константное выражение из знака минус и следующей за ним константы. Например, -12, -0хС.

Каждой целой константе присваивается тип, определяющий преобразования, которые должны быть выполнены, если константа используется в выражении. Тип константы зависит от ее значения и определяется следующим образом:

- десятичные константы рассматриваются как величины со знаком, и им присваивается тип `int` или `long` в соответствии с размером константы (типы данных рассматриваются в главе 2). Если константа меньше 32768, то ей присваивается тип `int`, в противном случае — тип `long`;

- восьмеричным и шестнадцатеричным константам присваивается тип `int`, `unsigned int`, `long` или `unsigned long` в зависимости от размера константы.

Диапазоны величин восьмеричных и шестнадцатеричных констант, представленных соответствующими типами на машине, где слово имеет длину 16 бит, приведены в табл. 1.5.

Таблица 1.5

Диапазон восьмеричных и шестнадцатеричных констант

Диапазон шестнадцатеричных констант	Диапазон восьмеричных констант	Тип
0x0–0x7FFF	0–077777	<code>int</code>
0x8000–0xFFFF	0100000–0177777	<code>unsigned int</code>
0x10000–0x7FFFFFFF	0200000–017777777777	<code>long</code>
0x80000000–0xFFFFFFFF	020000000000–037777777777	<code>unsigned long</code>

Чтобы определить для любой целой константы тип `long`, достаточно в конец константы приписать букву "l" или "L". Например, 5l, 010L, 0x1fl.

Константа с плавающей точкой — десятичное число, представляемое в виде действительной величины с десятичной точкой или экспонентой. Формат:

[цифры] [.цифры] [E|e [+|-] цифры]

Величина действительного числа включает целую и дробную части и экспоненту.

Цифры — это одна или более десятичных цифр (от 0 до 9), а "E" или "e" — символ экспоненты. В константе с плавающей точкой могут быть опущены либо целая, либо дробная части, но не обе сразу. Десятичная точка может быть опущена лишь тогда, когда задана экспонента. Пробельные символы не должны разделять цифры и символы константы.

Константы с плавающей точкой всегда представляют положитель-

ные величины удвоенной точности (имеют тип `double`). Для определения отрицательной величины необходимо формировать константное выражение, состоящее из знака минус с последующей константой.

Примеры констант с плавающей точкой и константных выражений: 115.75, 1.5E-2, -0.025, .75, .07e3, 2E-3.

Символьная константа — это символ, заключенный в апострофы. Так как управляющая последовательность рассматривается как одиночный символ, допустимо использовать ее в символьных константах. Значением символьной константы является числовой код символа.

Примеры символьных констант:

' ' — пробел;

'б' — буква "б";

'\n' — символ новой строки;

'\\' — обратная дробная черта.

Символьные константы имеют тип `int` и при преобразовании типов расширяются знаком.

Строковый литерал — это последовательность символов, заключенная в кавычки ("").

Например:

"Это строка символов"

Все непечатные символы, кавычки (""), обратная дробная черта (\) и символ новой строки в строковом литерале представляются соответствующими управляющими последовательностями. Каждая управляющая последовательность рассматривается как отдельный символ. Символы строкового литерала сохраняются в непрерывной области памяти. В конец каждого строкового литерала компилятором добавляется нулевой символ, представляемый управляющей последовательностью `\0`.

Строковые литералы имеют тип `char[ ]`. Это означает, что строка — это массив символов. Число элементов массива равно числу символов в строковом литерале плюс 1, так как нулевой символ также является элементом массива.

Все строковые литералы, в том числе и идентичные, в программе рассматриваются как различные объекты.

Строковые литералы могут располагаться на нескольких строках. Традиционным способом формирования таких литералов является использование обратной дробной черты с последующим нажатием клавиши ВВОД. Обратная черта с символом новой строки игнорируется компилятором, и следующая строка считается продолжением текущей.

Например, литерал

"Это очень длин\
ная строка"

идентичен литералу

"Это очень длинная строка"

Кроме того, два или более строковых литерала, разделенных только пробельными символами, также рассматриваются как одна строка. Такое сцепление строк можно использовать в любом месте программы вместо использования обратной дробной черты и символа новой строки для формирования строковых литералов, занимающих более одной строки.

Идентификаторы

Идентификаторы — это имена переменных, типов, функций и меток, используемых в программе. Идентификатор состоит из последовательности букв, цифр и символов подчеркивания (_). Первым символом любого идентификатора должна быть буква или символ подчеркивания. Компилятор Си рассматривает прописные и строчные буквы как различные символы. Например, идентификаторы `TEMP`, `temp`, `Tempr` считаются различными.

Компилятор допускает любое количество символов в идентификаторе, но значащими для компилятора являются лишь первые 31 символ. Идентификатор создается объявлением соответствующей ему переменной, типа или функции. После этого его можно использовать в последующих операторах программы. При выборе идентификаторов необходимо соблюдать некоторые правила:

- нельзя использовать идентификаторы, которые совпадают по написанию с ключевыми словами;
- следует быть осторожным при использовании символов подчеркивания в качестве первых символов идентификаторов, поскольку такие идентификаторы могут совпадать с именами системных программ и переменных. Не гарантируется, что программы, использующие имена с ведущими символами подчеркивания, будут переносимы;
- при создании внешних идентификаторов необходимо учитывать ограничения, накладываемые на количество и тип символов для глобальных имен редактором связей операционной системы.

Ключевые слова

Ключевые слова — это предопределенные идентификаторы, которые имеют специальное значение для компилятора Си. Их можно использовать только так, как они определены.

Список ключевых слов языка Си:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Кроме того, в описываемой реализации зарезервированными словами являются также следующие слова, представляющие расширение проекта стандарта АНИС:

<u>_asm</u>	fortran	near
cdecl	huge	pascal
far	interrupt	

Ключевые слова `far`, `huge` и `near` позволяют определить зависимые от архитектуры процессора размеры указателей и описываются в главе 9. Ключевые слова `_asm`, `cdecl`, `fortran`, `pascal` служат для организации связи с программами на других языках или оформления функции специальным образом (`interrupt`) и рассматриваются в главе 10. Ключевое слово `volatile` зарезервировано для будущего использования.

Ключевые слова нельзя использовать как имена переменных, типов, функций, меток.

Комментарии

Комментарий — это последовательность символов, которая воспринимается компилятором как отдельный пробельный символ и игнорируется. Формат:

/ символы */*

Комбинации символов `/*` и `*/` ограничивают комментарии. Символы между ограничителями комментария могут быть любой последовательностью символов, включая символы новой строки, но исключая комбинацию `*/`, обозначающую конец комментария. Комментарии могут занимать более одной строки, но не могут быть вложенными. Комментарии допускаются везде, где разрешены пробельные символы. Поэтому комментарии не могут появляться внутри лексем.

Символы комментария компилятором игнорируются, поэтому использование ключевых слов в комментариях не приводит ни к какому действию.

Примеры правильного использования комментариев:

/ Это комментарий */*

*/** Использование оператора while ***/*

Пример неправильного использования комментариев:

/ Недопустимы /* вложенные */ комментарии */*

Глава 2

ТИПЫ ДАННЫХ. ОБЪЯВЛЕНИЯ

В программе на языке Си все переменные перед использованием должны быть объявлены. Объявление переменной имеет следующий формат:

[спецификатор-класса-памяти] спецификатор-типа
описатель [=инициатор] [,описатель [=инициатор]]...

Описатель — это идентификатор, определяющий имя объявляемой переменной, который может быть определен с квадратными скобками, круглыми скобками или звездочкой.

Спецификатор-типа — одно или несколько ключевых слов, определяющие тип объявляемой переменной. Язык Си определяет стандартный набор основных типов данных, используя которые можно определять производные типы.

Инициатор задает начальное значение или список начальных значений, присваиваемых переменной при объявлении.

Четыре ключевых слова: *auto*, *extern*, *register* и *static* определяют *класс-памяти*, или каким образом для объявляемой переменной распределяется память и из каких частей программы можно ссылаться на нее. Если ключевое слово, определяющее класс памяти, опущено, то класс памяти определяется по контексту (подробнее в главе 6).

2.1. КАТЕГОРИИ ТИПОВ ДАННЫХ

Все типы данных можно разделить на две категории: скалярные и составные (рис. 2).



Рис. 2. Типы данных

Ключевые слова, используемые при объявлении основных типов данных, перечислены ниже.

Ключевые слова для определения основных типов данных:

Целые типы

char
int
short
long
signed
unsigned

Плавающие типы

float
double
long double

Составные типы данных "структура", "смесь", "перечисление" также определяются ключевыми словами соответственно `struct`, `union` и `enum`. Указатели и массивы определяются с помощью специальных синтаксических форм (п. 2.4 и 2.6).

Данное любого типа может быть объявлено как немодифицируемое. Для этого в спецификаторе типа достаточно указать ключевое слово `const`. Объекты с типом `const` представляют собой данные, используемые только для чтения. Если спецификатор типа состоит только из слова `const`, тип объявляемого объекта эквивалентен типу `const int`. Если с ключевым словом `const` объявляется данное составного типа, это означает, что каждый элемент такого данного является немодифицируемым.

Пример.

```
const float pi=3.14
```

Переменная `pi` имеет значение 3.14 и не может быть изменена.

Ключевое слово `void` означает отсутствие типа и позволяет отложить определение типа. Подробнее тип `void` рассматривается в связи с объявлением указателей и функций.

2.2. ЦЕЛЫЕ ТИПЫ ДАННЫХ

Различные ключевые слова, используемые для определения целых типов данных, определяют диапазон значений и размер памяти, распределяемой под переменные (табл. 2.1). От этого зависят машинные команды, которые будут использоваться для обработки данных.

Ключевые слова `signed` и `unsigned` необязательны. Они указывают, как интерпретируется нулевой бит переменной. Если указано ключевое слово `unsigned`, то нулевой бит рассматривается как часть числа. Если же указано ключевое слово `signed`, то нулевой бит интерпретируется как знак. По умолчанию все переменные целого типа считаются `signed`. Ключевые слова `signed` и `unsigned` могут предшествовать любому целому типу. Они могут употребляться и самостоятельно при определении переменной. В этом случае они рассматриваются соответственно как `signed int` и `unsigned int`.

Таблица 2.1

Размер памяти и диапазон значений целых типов

Тип	Размер памяти, байт	Диапазон значений
char	1	от -128 до 127
int	Зависит от реализации *	
short	2	от -32 768 до 32 767
long	4	от -2 147 483 648 до 2 147 483 647
unsigned char	1	от 0 до 255
unsigned int	Зависит от реализации *	
unsigned short	2	от 0 до 65535
unsigned long	4	от 0 до 4 294 967 295

* В данной реализации 2 байта.

Тип `char` используется для представления символа из множества представимых символов. Значением объекта типа `char` является код в наборе кодов ПЭВМ, соответствующий данному символу. Так как тип `char` по умолчанию интерпретируется как однобайтовая целая величина со знаком, диапазон значений переменной `char` от минус 128 до 127, хотя только значения от 0 до 127 имеют символьные эквиваленты. Аналогично переменная типа `unsigned char` может запоминать значения в диапазоне от 0 до 255.

Примечания.

- 1. С помощью режима компилятора можно изменить режим умолчания с `signed char` на `unsigned char`.
- 2. При необходимости использования в качестве значений переменной `char` русских букв надо объявлять переменную как `unsigned char`, так как коды русских букв превышают 127.

Заметим, что представление в памяти и диапазон значений для типов `int` и `unsigned int` не определены в языке Си. По умолчанию размер `int` (со знаком и без знака) соответствует размеру целого на данной машине. Например, на 16-разрядной машине тип `int` всегда 16 бит, или 2 байта. На 32-разрядной машине тип `int` всегда 32 бита, или 4 байта. Таким образом, тип `int` эквивалентен типам `short int` или `long int` в зависимости от реализации. Аналогично тип `unsigned int` эквивалентен типам `unsigned short` или `unsigned long`.

Спецификаторы типов `int` и `unsigned int` широко используются в Си-программах, поскольку они позволяют наиболее эффективно манипулировать целыми величинами на данной машине. Однако размер

типов `int` и `unsigned int` переменный, поэтому программы, зависящие от спецификации размера `int` и `unsigned int`, могут быть непереносимы. Переносимость программ можно улучшить путем включения выражений с операцией `sizeof`.

Диапазон значений величин — это интервал целых величин от минимального до максимального, который может быть представлен в заданном числе битов. Тем не менее не всегда возможно использовать максимальное или минимальное значение для константы данного типа в выражении. Например, константное выражение `(-32 768)` состоит из знака арифметического отрицания `(-)`, предшествующего величине константы `32 768`. Так как `32 768` — это слишком большое значение для типа `short`, то оно задается типом `long` и, следовательно, константное выражение `(-32 768)` будет иметь тип `long`. Величина `(-32 768)` может быть представлена как `short` только путем преобразования ее к типу `short`. Информация не будет потеряна при преобразовании типа, поскольку `(-32 768)` может быть представлено двумя байтами памяти. Аналогично такая величина как `65 000` может быть представлена как `unsigned short` только путем преобразования ее к типу `unsigned short` или заданием величины в восьмеричном или шестнадцатеричном представлении. Величина `65 000` в десятичном представлении рассматривается как константа со знаком и задается типом `long`, так как `65 000` не соответствует типу `short`. Эта величина типа `long` может быть преобразована к типу `unsigned short` без потери информации, так как `65 000` можно разместить в двух байтах памяти.

Восьмеричные и шестнадцатеричные константы могут быть типа `signed` или `unsigned` в зависимости от их размера. Однако метод, используемый для назначения типов этим константам, гарантирует, что они всегда будут преобразованы к целому без знака.

Пример.

```
main()
{
    int a, b, c, d;
    short s, t, u;
    long l, m, n;
    unsigned int ui, uj, uk;
    a = 17;
    b = -197;
    c = 0xa7c;
    d = 01777;
    /* Печать переменных типа int как десятичных целых */
    printf("a=%d\t b=%d\t c=%d\t d=%d\n", a, b, c, d);
    s = 32;
    t = -32767;
    u = -1;
    /* Печать переменных типа short */
    printf("s= %d\t c= %d\t u= %d\n", s, t, u);
```



```

l = -15L;
m = -455;
n = 156765341;
    /* Печать переменных типа long */
printf("l = %ld\t m = %ld\t n = %ld\n", l, m, n);
ui = 157;
uj = 0xff;
uk = -15;
    /* Печать целых без знака в десятичном формате */
printf("ui = %u\t uj = %u\t uk = %u\n", ui, uj, uk);
    /* В восьмеричном и шестнадцатеричном форматах */
printf("uj(восьмерич) = %o\t uj(шестн) = %x\n", uj, uj);
}

```

Результат выполнения программы

```

a = 17      b = -197      c = 2684      d = 1023
s = 32      t = -32767    u = -1
l = -15     m = -455      n = 156765341
ui = 157    uj = 255      uk = 65521
uj(восьмерич) = 377    uj(шестн) = ff

```

В этом и других примерах для выдачи результатов используется функция `printf`, выполняющая форматизованный вывод. Эта функция подробно описана в главе 11.

2.3. ПЛАВАЮЩИЕ ТИПЫ ДАННЫХ

Для объявления переменных плавающего типа используются ключевые слова `float`, `double` и `long double`.

Все числа с плавающей точкой представляются в формате IEEE. Величины типа `float` занимают 4 байта. Из них 1 бит отводится для знака, 8 бит для избыточной экспоненты и 23 бита для мантиссы. Мантисса — это число между 1.0 и 2.0. Так как старший бит мантиссы всегда 1, он не запоминается в памяти. Такое представление дает диапазон значений приблизительно от $3.14E-38$ до $3.14E+38$.

Величины типа `double` занимают 8 байт памяти. Их формат аналогичен формату `float`. Биты памяти распределяются следующим образом: 1 бит для знака, 11 бит для экспоненты и 52 бита для мантиссы. С учетом опущенного старшего бита мантиссы диапазон значений получается приблизительно от $1.7E-308$ до $1.7E+308$.

Тип `long double` семантически эквивалентен типу `double`.

Пример.

```

main()
{
float f, g, h;
double w, v;
f = 0.1245e2;
g = -0.123E-3;
h = 0.00013;

```



```

w = 2.0;
v = -271.2164982E17;
printf ("f = %f\t g = %f\t h = %f\n", f, g, h);
printf ("w = %f\t v = %f\n", w, v);
printf ("w = %e\t v = %e\n", w, v);
}

```

Результат выполнения программы

```

f = 12.450000      g = -0.000123      h = 0.000130
w = 2.000000      v = -271216498200000000000.000000
w = 2.000000e+000  v = -2.712165e+019

```

2.4. УКАЗАТЕЛИ

Указатель — это адрес памяти, распределенной для другой переменной. Если переменная объявлена как указатель, она может хранить адрес и таким образом указывать на другое значение. При объявлении переменной типа указатель необходимо указать тип данных, адреса которых будет содержать переменная, и имя указателя с предшествующей звездочкой. Формат объявления указателя:

спецификатор-типа * [*модификатор*] *идентификатор*;

Спецификатор-типа задает тип объекта и может быть любого основного типа, типа структуры, смеси. Указатели могут указывать также на функции, массивы, другие указатели.

Задавая вместо спецификатора-типа ключевое слово `void`, можно как бы отсрочить спецификацию типа, на который ссылается указатель. Переменная, объявленная как указатель на тип `void`, может быть использована для ссылки на объект любого типа. Однако для того чтобы выполнять операции над такими указателями или над объектами, на которые такие указатели ссылаются, тип, на который они ссылаются, должен быть явно задан в каждой операции. Такое определение типа может быть выполнено с помощью операции приведения типов.

В качестве *модификатора* при объявлении указателя могут выступать ключевые слова `const`, `near`, `far`, `huge`.

Ключевое слово `const` объявляет, что указатель не может быть изменен в программе.

Размер переменной, объявленной как указатель, зависит от конфигурации машины и от модели памяти, для которой будет компилироваться программа. Указатели на различные типы данных не обязательно имеют одинаковую длину.

Для модификации размера указателя можно использовать ключевые слова `near`, `far` и `huge` (подробнее в главе 9).

Примеры.

```

/***** Пример 1 *****/
unsigned int * point;

```


Переменная `point` представляет собой указатель на тип целых чисел без знака.

```
/****** Пример 2 *****/  
double *x;
```

Переменная `"x"` является указателем на тип данных с плавающей точкой удвоенной длины.

```
/****** Пример 3 *****/  
char *text;
```

Объявляется указатель с именем `text`, который указывает на переменную типа `char`.

```
/****** Пример 4 (указатель на тип void) *****/  
float number;  
void *address;  
address = &number;  
(float *) address++;
```

Переменная `address` объявлена как указатель на объект любого типа. Поэтому ей можно присвоить адрес переменной, например, плавающего типа. Однако ни одна операция не может быть выполнена над указателем до тех пор, пока не будет явно задан тип данных, на который он указывает. Это можно сделать, используя операцию приведения типа `(float *)` для преобразования `address` к указателю на тип `float`, и затем произвести увеличение значения адреса.

```
/****** Пример 5 (использование const) *****/  
const * add;  
unsigned char * const w = &obj;
```

Переменная `add` объявлена как указатель на константное выражение, т.е. значение указателя может изменяться в программе, а величина, на которую он указывает, — нет.

Переменная `"w"` объявляется как константный указатель на данное типа `unsigned char`. Это значит, что на протяжении всей программы `"w"` будет указывать на одну и ту же ячейку памяти. Содержимое же этой ячейки может быть изменено.

2.5. ПЕРЕЧИСЛЕНИЯ

Если переменная может принимать лишь одно значение из определенного множества значений, рекомендуется использовать спецификацию типа "перечисление" для определения возможных значений переменной.

Объявление перечисления начинается с ключевого слова `enum` и имеет два формата представления.

Формат 1:

```
enum [имя-типа-перечисления] {список-перечисления}
    [описатель[,описатель]...];
```

Формат 2:

```
enum имя-типа-перечисления описатель [,описатель]...;
```

Объявление перечисления задает имя переменной перечисления и определяет список именованных констант, называемый *списком-перечисления*. Значением каждого имени списка является целое число. Переменная перечисления принимает значение одной из именованных констант списка. Именованные константы списка имеют тип `int`. Таким образом, память, соответствующая переменной перечисления, — это память, необходимая для размещения значения типа `int`.

Переменные типа `enum` во всех случаях рассматриваются как переменные типа `int`. Они могут использоваться в индексных выражениях и как операнды арифметических операций и операций отношения.

В формате 1 имена и значения перечисления задаются в списке перечисления. Необязательное *имя-типа-перечисления*, — это идентификатор, который именует тип перечисления, определенного списком перечисления. *Описатель* именует переменную перечисления. В объявлении может быть задана более чем одна переменная перечисления.

Список-перечисления имеет следующий формат:

```
идентификатор [=константное-выражение]
    [,идентификатор [=константное-выражение]]...
```

Каждый *идентификатор* именует элементы перечисления. Все идентификаторы в списке `enum` должны быть уникальны. По умолчанию первому идентификатору соответствует значение 0, следующему идентификатору — значение 1 и т.д. Имя константы перечисления эквивалентно ее значению.

Необязательная фраза "*=константное-выражение*" переопределяет последовательность значений, подразумеваемых по умолчанию. Идентификатор, связанный с константным выражением, принимает значение, задаваемое этим константным выражением. Константное выражение должно иметь тип `int` и может быть отрицательным. Следующему идентификатору в списке присваивается значение, равное константному выражению плюс 1, если оно явно не задается другой величиной.

Правила использования элементов перечисления.

1. Перечисление может содержать повторяющиеся значения.

2. Идентификаторы в списке перечисления должны быть отличны от всех других идентификаторов в той же области видимости, включая имена обычных переменных и идентификаторы из других списков перечислений.

3. Имена типов перечислений должны быть отличны от других имен типов перечислений, структур и смесей в той же области видимости.

4. Запятая может следовать за последним элементом списка перечисления.

Пример.

```
enum НЕДЕЛЯ {  
    суббота = 0,  
    воскресенье = 0,  
    понедельник,  
    вторник,  
    среда,  
    четверг,  
    пятница  
}    раб_неделя;
```

В формате 2 используется имя типа перечисления для ссылки на тип перечисления, определенный где-то в другом месте. Имя типа перечисления должно относиться к уже определенному типу перечисления в пределах текущей области видимости. Так как тип перечисления объявлен где-то в другом месте, список перечисления не представлен в объявлении. Например,

```
enum НЕДЕЛЯ раб_неделя;
```

В объявлениях указателей на перечисления и объявлениях typedef для типов перечисления можно использовать имя типа перечисления до того, как данный тип перечисления определен. Однако определение перечисления должно предшествовать любому действительному использованию указателя или объявления typedef.

Объявление без последующего списка описателей просто описывает шаблон перечисления.

Пример программы, использующей данные типа перечисление:

```
#define NUMDAYS 7  
enum DAYS  
{  
    воскресенье, понедельник, вторник, среда, четверг, пятница, суббота  
};  
main ()  
{  
    enum DAYS day1, day2, day3, day_before(), day_after();  
    void print_day();  
    day1 = воскресенье;  
    day2 = day_after(day1);  
    day3 = day_before(day1);  
    printf("Если сегодня "), print_day(day1);  
    printf(", то завтра будет "), print_day(day2);  
    printf(", \nна вчера была "), print_day(day3);  
    printf(".\n");  
}  
enum DAYS day_after(day)
```



```
enum DAYS day;
{
    return((enum DAYS)((day + 1) % NUMDAYS));
}
enum DAYS day_before(day)
enum DAYS day;
{
    int prev;
    prev = (day - 1) % NUMDAYS;
    return(prev < 0 ? (NUMDAYS - 1) : prev;
}
void print_day(day)
enum DAYS day;
{
    int day_i = day;
    static char *days[] =
    {
        "воскресенье",
        "понедельник",
        "вторник",
        "среда",
        "четверг",
        "пятница",
        "суббота"
    };
    if (day_i < 0 || day_i >= NUMDAYS)
        printf("ОШИБКА");
    else
        printf("%s", days[day_i]);
}
```

Результат выполнения программы

Если сегодня воскресенье, то завтра будет понедельник,
а вчера была суббота.

2.6. МАССИВЫ

Массив — это группа элементов одинакового типа. Из объявления массива компилятор должен получить информацию о типе элементов массива и количестве элементов.

Объявление массива имеет два формата.

Формат 1:

спецификатор-типа *описатель* [*константное-выражение*]...;

Формат 2:

спецификатор-типа *описатель* [];

Описатель в форматах именуется переменной. Квадратные скобки указывают на то, что объявляется массив.

Спецификатор-типа задает тип элементов объявляемого массива. Элементами массива не могут быть функции и элементы типа void. *Константное-выражение* в квадратных скобках задает количество

элементов в массиве. Константное выражение при объявлении массива может быть опущено в следующих случаях:

- при объявлении массив инициализируется;
- массив объявлен как формальный параметр;
- массив объявлен как ссылка на массив, явно определен в другом файле.

Допускается использование многомерных массивов. Они объявляются путем задания списка константных выражений в квадратных скобках, следующего за описателем массива.

Каждое константное выражение в квадратных скобках определяет число элементов в данном измерении массива, так что объявление двумерного массива содержит два константных выражения, трехмерного — три и т.д.

Память, распределяемая под массив, равна памяти для размещения всех его элементов. Элементы массива с первого до последнего запоминаются в последовательных возрастающих адресах памяти. Между элементами массива в памяти разрывы отсутствуют. Двумерные массивы запоминаются в памяти построчно. Многомерные массивы запоминаются в памяти таким образом, что, если переходить от элемента к элементу последовательно, последний индекс изменяется быстрее всего. Например, двумерный массив, объявленный как

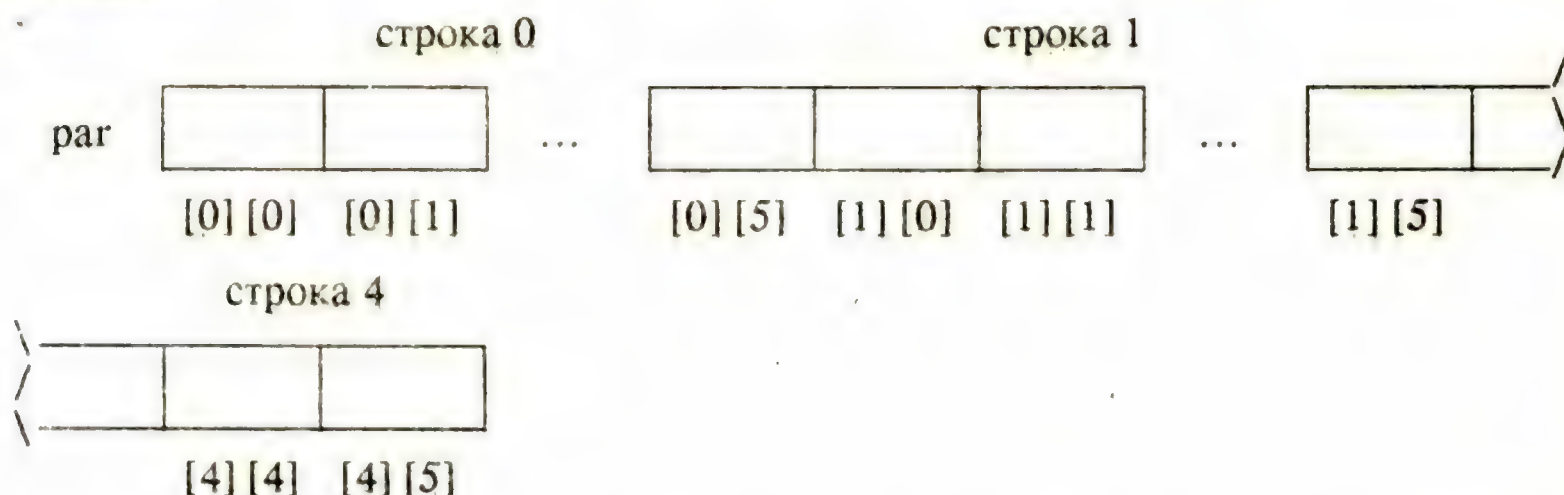
```
int par [5][6];
```

программист представляет в виде матрицы (элементы массива нумеруются в языке Си с нуля):

Столбцов 6

		[0][0]	[0][1]	[0][2]	[0][3]	[0][4]	[0][5]
Строк 5	par[0]						
	par[1]	[1][0]	[1][1]	[1][2]	[1][3]	[1][4]	[1][5]
	par[2]	[2][0]	[2][1]	[2][2]	[2][3]	[2][4]	[2][5]
	par[3]	[3][0]	[3][1]	[3][2]	[3][3]	[3][4]	[3][5]
	par[4]	[4][0]	[4][1]	[4][2]	[4][3]	[4][4]	[4][5]

В памяти элементы этого массива будут размещены следующим образом:



При ссылке на отдельный элемент массива используется индексное выражение (глава 3).

Примеры объявления массивов:

```
double value [20]; /* Массив из 20 элементов типа double */
int tablica [10] [10]; /* Двумерный массив элементов типа int */
```

Так как строка в языке Си определяется как массив символов, последним элементом которого является символ '\0', то следующее объявление также является примером объявления массива:

```
char string[] = "Это массив";
```

2.7. СТРУКТУРЫ

В отличие от массивов, все элементы которых должны быть одного типа, структуры объединяют в одной переменной элементы разных типов.

Объявление структуры начинается с ключевого слова `struct` и имеет 2 формата.

Формат 1:

```
struct [имя-типа-структуры] {список-объявлений-элементов}
    [описатель[,описатель]...];
```

Формат 2:

```
struct имя-типа-структуры описатель[,описатель]...;
```

Объявление структуры задает имя структурной переменной и последовательность переменных величин, называемых элементами структуры, которые могут иметь различные типы. Переменная этого структурного типа содержит целую последовательность величин, определяемую этим типом.

В формате 1 *список-объявлений-элементов* задает типы и имена элементов структуры. *Имя-типа-структуры* — это идентификатор,

который именуется тип структуры, определяемый списком объявлений элементов.

В формате 2 используется имя типа структуры для ссылки на тип структуры, определенный где-то в другом месте в области видимости. Поэтому список объявлений элементов в этом формате отсутствует.

В объявлениях указателей на структуры и объявлениях `typedef` может использоваться имя типа структуры до того, как данный тип структуры будет определен. Однако определение структуры должно предшествовать любому действительному использованию указателя или объекта, объявленного через `typedef`.

В обоих форматах *описатель* именуется переменную структурного типа. В качестве описателя может быть имя переменной, модифицированной в указатель, массив или функцию.

Указатель рассматривается как указатель на структуру, массив — как массив структур, а функция — как функция, возвращающая структуру.

Если задано имя типа структуры без последующего списка переменных, то такое объявление вводит шаблон структуры.

Имена типов структур должны отличаться от других имен типов структур, смесей и перечислений в той же области видимости.

Список-объявлений-элементов содержит одно или несколько объявлений переменных или *полей-битов*. Каждая переменная, объявленная в этом списке, называется элементом структурного типа. Объявления переменных списка имеют такой же синтаксис, что и другие объявления переменных, рассматриваемые в этой главе, за исключением того, что объявления не могут содержать спецификаторов класса памяти или списка начальных значений. Элементы структуры могут быть любого основного типа, массивом, указателем, смесью или структурой.

Пример.

```
struct УЧИТЕЛЬ {  
    char фио[40];  
    int таб_ном;  
    float оклад;  
    } список[20], *ук;  
struct УЧИТЕЛЬ школа();
```

В этом примере объявляется массив структур типа `УЧИТЕЛЬ` из 20 элементов, указатель на структуру типа `УЧИТЕЛЬ` и функция, возвращающая структуру типа `УЧИТЕЛЬ`.

Структура не может содержать в качестве элемента структуру такого же типа, но может включать указатель на структуру этого типа

при условии, что в объявлении структуры указано имя типа. Это позволяет создавать связанные списки структур. Например:

```
struct tree {
    int number;
    struct tree * left;
    struct tree * right;
};
```

В языке Си есть возможность работать с битовыми полями. Битовое поле (поле-битов) может быть объявлено только как элемент структуры. Объявление *поля-битов* имеет следующий формат:

спецификатор-типа [*идентификатор*]:*константное-выражение*;

Константное-выражение задает число битов в поле и должно быть неотрицательной целой величиной. *Спецификатор-типа* должен задавать целый тип (signed или unsigned). Не допускаются массивы полей битов, указатели на поля битов и функции, возвращающие поля битов. Необязательный *идентификатор* именует поле битов. Неименованные поля используются для выравнивания. Неименованное поле с нулевым размером означает, что память для следующего элемента объявления будет начинаться на границе int.

Для полей битов можно указывать типы char и long (signed и unsigned). Неименованные поля битов с типами long или char (signed или unsigned) служат для выравнивания на соответствующую границу основного типа. Поля битов с типом signed поддерживаются только синтаксически. Во всех преобразованиях поля битов с типом signed будут рассматриваться как unsigned.

Пример.

```
#include <bios.h>
main()
{
    struct eq {
        unsigned flop_boot : 1; /* дискеты */
        unsigned s8087 : 1; /* сопроцессор */
        unsigned mram : 2; /* оперативная память в блоках */
        unsigned video_mode : 2; /* режим видеомонитора */
        unsigned numflop : 2; /* кол-во дисководов:
                                00 — 1, 01 — 2; 10 — 3, 11 — 4 */
        unsigned dma : 1; /* DMA */
        unsigned ports : 3; /* кол-во портов СТЫК С2 */
        unsigned game_adap : 1; /* игровой адаптер */
        unsigned : 1;
        unsigned numprint : 2; /* кол-во печатающих устройств */
    } eq;
    unsigned int *i;
    i = (unsigned int *) &eq;
    *i = _bios_equiplist();
    printf("КОЛИЧЕСТВО ДИСКОВОДОВ — %d\n", eq.numflop + 1);
    printf("КОЛИЧЕСТВО ПОРТОВ = %d\n", eq.ports);
}
```



```
if (eq.s8087)
    printf("СОПРОЦЕССОР ЕСТЬ\n");
else
    printf("СОПРОЦЕССОРА НЕТ\n");
}
```

Результат выполнения программы
КОЛИЧЕСТВО ДИСКОВОДОВ — 2
КОЛИЧЕСТВО ПОРТОВ — 2
СОПРОЦЕССОР ЕСТЬ

В этом примере элементами структурной переменной `eq` являются поля битов, описывающие список установленного оборудования в BIOS. Этот список получается с помощью стандартной функции библиотеки СП Си `_bios_equiplist`. Далее элементы структуры `eq` используются в арифметическом выражении (`eq.numflor + 1`), в условном выражении оператора `if` как целые переменные.

Поля битов не могут располагаться на пересечении границ объявленных для них типов. Например, поле, объявленное с типом `unsigned int`, размещается в пространстве, оставшемся от предыдущего поля с типом `unsigned int`, либо начиная с новой целой границы.

Идентификаторы в списке объявлений элементов должны быть уникальны. Однако идентификаторы элементов разных структур могут совпадать между собой и с идентификаторами обычных переменных.

Элементы структуры запоминаются последовательно в том же порядке, в котором они объявляются: первому элементу соответствует самый младший адрес памяти, а последнему — самый старший. Память каждому элементу выделяется на границе, соответствующей его типу. Поэтому в памяти могут появиться неименованные участки ("дыры") между соседними элементами. Вследствие этого размер переменной структурного типа не всегда равен сумме размеров ее элементов. Для определения действительного размера структурной переменной надо использовать операцию `sizeof`.

В компиляторах СП Си (как в компиляторе среды Си, так и в оптимизирующем компиляторе) имеется режим компиляции, позволяющий управлять упаковкой элементов структуры в памяти.

2.8. СМЕСИ

Значения различных типов можно хранить в одной и той же области памяти (но не одновременно). Для этого используется тип "смесь". Фактически смесь — это структура, все элементы которой имеют нулевое смещение, а сама структура достаточно велика, чтобы вместить самый большой элемент, и выровнена так, чтобы можно было работать с любым элементом.

Объявление смеси имеет тот же синтаксис, что и объявление струк-

туры, за исключением того, что оно начинается с ключевого слова `union` вместо ключевого слова `struct`. Для объявления смеси и структуры действуют одни и те же правила, только в смеси не допускаются элементы типа полей битов.

Объявление смеси имеет два формата.

Формат 1:

```
union [имя-типа-смеси] {список-объявлений-элементов}
    [описатель[,описатель]...];
```

Формат 2:

```
union имя-типа-смеси описатель[,описатель]...;
```

Пример.

```
union num {
    int      integer;
    float    float_num;
    double   complex;
} number;
```

Память, которая соответствует переменной типа смеси, определяется величиной, необходимой для размещения наибольшего элемента смеси. Когда используется меньший элемент смеси, то переменная типа смеси может содержать неиспользуемую память. Все элементы смеси хранятся в одной и той же области, начиная с одного и того же адреса.

В каждый момент времени в переменной типа смесь хранится только одно значение. Помнить о том, какой элемент хранится в смеси в данный момент, должен сам программист. Если в переменную было загружено значение одного типа, а используется другое, это приведет к неправильным результатам. Однако часто смеси применяются в случаях, когда надо выделить часть целой переменной или загрузить целое значение в структуру, описанную с полями битов.

Пример.

```
#include <stdio.h>
/* Программа выдачи двоичного кода введенного символа */
/* Для окончания ввести символ к */
struct byte {
    unsigned a: 1;
    unsigned b: 1;
    unsigned c: 1;
    unsigned d: 1;
    unsigned e: 1;
    unsigned f: 1;
    unsigned g: 1;
    unsigned h: 1;
};
union symb {
    unsigned char ch;
    struct byte bit;
```



```

        } symbol;
void convert(union symb p); /* Функция выдачи двоичного эквивалента */
main()
{
    do {
        symbol.ch = getche(); /* Чтение с отображением очередного символа */
        printf(" : ");
        convert(symbol);
        } while (symbol.ch != 'к');
    }
void convert(union symb p)
{
    if (p.bit.h) printf("1");
        else printf("0");
    if (p.bit.g) printf("1");
        else printf("0");
    if (p.bit.f) printf("1");
        else printf("0");
    if (p.bit.e) printf("1");
        else printf("0");
    if (p.bit.d) printf("1");
        else printf("0");
    if (p.bit.c) printf("1");
        else printf("0");
    if (p.bit.b) printf("1");
        else printf("0");
    if (p.bit.a) printf("1");
        else printf("0");
    printf("\n");
}

```

Результат выполнения программы:

```

g : 01100111
f : 01100110
2 : 00110010
= : 00111101
: : 00111010
! : 00100001
\ : 01011100
л : 11011011
п : 11011111
Ж: 10110110
И: 10111000
б : 11010001
к : 11011010

```

2.9. ОБЪЯВЛЕНИЯ ОБЪЕКТОВ И ТИПОВ

Все переменные должны быть объявлены. Тип объявляемой переменной зависит от того, какое ключевое слово используется в качестве спецификатора типа и является ли описатель простым идентификатором или же комбинацией идентификатора с модификаторами, приведенными ниже.

Модификатор	Определяемый тип
*	Указатель
[]	Массив
()	Функция

При объявлении простой переменной, структуры, смеси или перечисления описатель — это просто идентификатор. Для объявления указателя, массива или функции идентификатор модифицируется соответственно звездочкой, квадратными скобками или круглыми скобками.

Кроме того, можно использовать одновременно более одного модификатора, что дает возможность создавать бесконечное множество *сложных описателей типов*. При использовании различных комбинаций модификаторов надо помнить, что некоторые комбинации недопустимы, а именно:

- элементами массива не могут быть функции;
- функции не могут возвращать массив или функцию.

При интерпретации сложных описателей квадратные и круглые скобки (справа от идентификатора) имеют приоритет перед звездочкой (слева от идентификатора). Квадратные или круглые скобки имеют один и тот же приоритет и связываются слева направо. Спецификатор типа рассматривается на последнем шаге, когда описатель уже полностью проинтерпретирован. Можно использовать круглые скобки, чтобы изменить порядок интерпретации на необходимый.

Для интерпретации сложных описателей предлагается простое правило, которое звучит как "изнутри наружу", из четырех шагов.

1. Начать с идентификатора и посмотреть вправо, есть ли квадратные или круглые скобки.

2. Если они есть, то проинтерпретировать эту часть описателя и затем посмотреть налево в поиске звездочки.

3. Если на любой стадии справа встретится закрывающая круглая скобка, то вначале необходимо применить все эти правила внутри круглых скобок, а затем продолжить интерпретацию.

4. Интерпретировать спецификатор типа.

Пример.

```
int      * (* comp[10]) () ;
6       5 3   1   2   4
```

В данном примере объявляется переменная `comp` (1) как массив из десяти (2) указателей (3) на функции (4), возвращающие указатели (5) на целые значения (6).

Кроме объявлений переменных различных типов, имеется возможность объявлять типы. Это можно сделать двумя способами:

- указать имя типа при объявлении структуры, смеси или перечисления, а затем использовать это имя в объявлениях переменных и функций в качестве ссылки на этот тип;
- для объявления типа использовать ключевое слово `typedef`.

Объявления типов структур, смесей и перечислений имеют тот же синтаксический формат, что и объявления переменных этих типов. Однако имеется несколько отличий:

1. В объявлении типа идентификатор переменной отсутствует, так как нет объявляемой переменной.

2. В объявлении типа требуется имя, оно именует тип структуры, смеси или перечисления.

3. В объявлении типа должен присутствовать список объявлений элементов или список перечисления. В объявлении типа не допускается сокращенная форма, в которой имя типа ссылается на тип, определенный где-то в другом месте.

Пример объявления типа структуры *anketa*:

```
struct anketa {
    char   fio[40];
    int    vozrast;
    float  oklad};
```

Объявление *typedef* определяет спецификатор типа для типа. Это объявление используется для того, чтобы создать более короткие или mnemonic имена для типов, уже определенных в языке Си или объявленных пользователем.

Формат:

typedef спецификатор-типа описатель [,описатель]...;

Идентификатор, объявленный с ключевым словом *typedef*, становится синонимом имени типа и может затем использоваться в качестве спецификатора типа при объявлении данных. Этот идентификатор может использоваться вместе с ключевым словом *const* для объявления немодифицируемого данного.

Любой тип может быть объявлен с *typedef*, включая типы указателя, функции и массива. Имя с ключевым словом *typedef* для типов указателя, структуры или смеси может быть объявлено прежде, чем эти типы будут определены, но в пределах видимости объявления.

*/***** Пример 1 *****/*

```
typedef double (*MATH) ();
```

MATH — новое имя типа, представляющего указатель на функцию, возвращающую значение типа *double*.

```
MATH tang;
```

tang — указатель на функцию, возвращающую значение типа *double*. Это эквивалентно объявлению:

```
double (*tang) ();
```

*/***** Пример 2 *****/*

```
typedef char FIO[40];
```

FIO — массив из сорока символов.

```
FIO person, * addr ;
```


Переменная `person` — массив символов, а `addr` — указатель на массив символов. Это эквивалентно объявлениям:

```
char person[40], *addr;
```

При объявлении переменных и типов мы использовали имя типа. Помимо этого, имена типов могут еще использоваться в трех контекстах: в списке формальных параметров в объявлениях функций, в операциях приведения типов и в операции `sizeof`.

Именами типов для основных типов, типов перечисления, структуры и смеси являются спецификаторы типов для этих типов.

Имена типов для типов указателя, массива и функции задаются следующим образом:

спецификатор-типа абстрактный-описатель

Абстрактный-описатель — это описатель без идентификатора, состоящий из одного или более модификаторов указателя, массива или функции. Модификатор указателя (*) всегда задается перед идентификатором в описателе, а модификаторы массива ([]) и функции (()) — после него. Таким образом, чтобы правильно интерпретировать абстрактный описатель, нужно начинать интерпретацию с подразумеваемого идентификатора.

Абстрактные описатели могут быть сложными. Скобки в сложном абстрактном описателе задают порядок интерпретации подобно тому, как это делалось при интерпретации сложных описателей в объявлениях.

Примечание. Не допускается абстрактный описатель, состоящий из пустых круглых скобок, поскольку это приводит к двусмысленности. В этом случае невозможно определить, находится ли подразумеваемый идентификатор внутри скобок (в таком случае это немодифицированный тип) или перед скобками (тогда это тип функции).

Спецификаторы типа, установленные посредством объявлений `typedef`, также рассматриваются как имена типов.

2.10. ИНИЦИАЛИЗАЦИЯ ДАННЫХ

При объявлении переменной можно присвоить начальное значение, присоединяя инициатор к описателю. Инициатор начинается с символа `=`.

Формат 1:

`= инициатор`

Формат 2:

`= { список-инициаторов }`

Формат 1 используется при инициализации переменных основных типов и указателей, а формат 2 — при инициализации составных объектов.

При инициализации переменных основных типов и указателей в качестве инициатора используется выражение.

Примеры.

```
char bell = '\007'; /* переменная bell инициализируется  
    символом '\007' */
```

```
const long megabyte=(1024*1024);
```

/ Переменная megabyte инициализируется константным выражением, после чего она не может быть изменена */*

При инициализации составных объектов в качестве инициатора используется список инициаторов, заключенных в фигурные скобки.

Список-инициаторов — это последовательность инициаторов, разделенных запятыми. Каждый инициатор в списке — это либо константное выражение, либо список инициаторов, т.е. заключенный в фигурные скобки список инициаторов может появиться внутри другого списка инициаторов.

Для каждого списка инициаторов значения константных выражений присваиваются в порядке следования элементов составной переменной. Если инициализируется смесь, то список инициаторов представляет собой единственное константное выражение. Величина константного выражения присваивается первому элементу смеси.

Если в списке инициаторов меньше значений, чем их имеется в составном типе, то оставшиеся элементы инициализируются нулем. Если число инициализирующих значений больше, чем требуется, имеет место ошибка.

Эти правила применяются к каждому вложенному списку инициаторов точно так же, как и ко всей конструкции в целом.

*/***** Пример 1 *****/*

```
static int board [2] [3]={1, 2, 3, 4, 5, 6};
```

Инициализируется двумерный массив board целых величин. Элементам массива присваиваются значения из списка построчно. Эта инициализация может быть проведена и иначе:

```
static int board [2] [3]={  
    {1, 2, 3},  
    {4, 5, 6}};
```

В этом варианте инициаторы элементов каждой строки заключены в фигурные скобки. Если при объявлении массива он инициализируется, то можно опускать размерность. В этом случае размерность массива будет вычисляться по количеству инициаторов. Тот же пример может быть записан следующим образом:

```
static int board [] []={  
    {1, 2, 3},  
    {4, 5, 6}};
```


Если же при инициализации указано меньше значений для строк, то оставшиеся элементы инициализируются нулем. Если объявлен массив

```
static board[2][3] = {
    {1, 2},
    {4}};
```

то элементы первой строки получают начальные значения 1,2,0 соответственно, а элементы второй строки — значения 4,0,0.

Когда инициализируются составные переменные, то нужно позаботиться о том, чтобы правильно использовать фигурные скобки и списки. В следующем примере показана более детальная интерпретация компилятором фигурных скобок.

```
/****** Пример 2 *****/
struct complex {
    float real;
    float imag;
} comp[2][3] = {
    {{ 1.0, 1.0},{2.0, 2.0},{3.0, 3.0}}, /* строка 1 */
    {{ 0.1, 0.1},{0.2, 0.2},{0.3, 0.3}} /* строка 2 */
};
```

В этом примере объявляется и инициализируется массив структур из двух строк и трех столбцов. Каждая структура состоит из двух элементов. Первая левая фигурная скобка говорит о начале составного инициатора. Вторая — о начале инициализации первой строки массива `comp`. Третья — о начале инициализации 1-го элемента первой строки (т.е. `comp[0][0]`). Последняя правая скобка в строке 1 сообщает компилятору об окончании инициализации первой строки массива. Аналогично строка 2 инициализирует вторую строку массива `comp`. Внешние фигурные скобки в строках 1 и 2 в примере необходимы. Если объявление будет сделано в виде

```
struct complex comp[2][3] = {
    {1.0, 1.0},{2.0, 2.0},{3.0, 3.0},
    {0.1, 0.1},{0.2, 0.2},{0.3, 0.3}}
};
```

то компилятор проинтерпретирует фигурные скобки следующим образом:

первая левая фигурная скобка — начало составного инициатора массива `comp`;

вторая левая фигурная скобка — начало инициализации первой строки массива (`comp[0]`). Значения 1.0, 1.0 назначаются двум элементам первой структуры. Первая правая фигурная скобка (после 1.0) указывает компилятору, что список инициаторов для строки массива окончен, и элементы оставшихся структур в строке `comp[0]` автоматически инициализируются нулем. Аналогично список {2.0, 2.0} инициализирует первую структуру в строке `comp[1]`, а оставшиеся структу-

ры массива установятся в нуль. На следующий список инициаторов компилятор выдает сообщение об ошибке.

При инициализации смеси задается значение первого элемента смеси в соответствии с его типом. Например:

```
union tab {  
    unsigned char  name [10];  
    int            tab;  
    } pers ={'А','н','т','о','н'};
```

Инициализируется переменная `pers`. Так как первый элемент смеси — массив, для его инициализации требуется список инициаторов в фигурных скобках. Первые пять элементов массива инициализируются значениями из списка, а остальные — нулем.

При инициализации массива символов в качестве инициатора можно использовать строковый литерал.

Например,

```
char строка[ ] = "привет";
```

инициализирует переменную "строка" как массив символов из семи элементов. Седьмым элементом является нулевой символ ('\0'), который завершает все строковые литералы.

Если задается размер массива, а строка длиннее, чем заданный размер, то лишние символы отбрасываются. Следующее объявление инициализирует переменную "строка" как пятиэлементный массив символов:

```
char строка[5] = "привет";
```

Только первые пять символов строки присваиваются массиву "строка". Символ "т" и нулевой символ отбрасываются.

Если строка короче, чем размер массива, то оставшиеся элементы массива заполняются нулем.

Таким образом, инициализация переменной `pers` типа смеси может иметь следующий вид:

```
union tab pers = "Антон";
```

Глава 3

ВЫРАЖЕНИЯ И ПРИСВАИВАНИЯ

Комбинация знаков операций и операндов, результатом которой является отдельное значение, называется *выражением*. Знаки операций определяют действия, которые должны быть выполнены над операндами. Каждый операнд в выражении в свою очередь является выражением, представляющим отдельное значение. Значение выражения

зависит от расположения знаков операций и круглых скобок в выражении, а также от старшинства операций.

В языке Си присваивания также являются выражениями, так как значением такого выражения является величина, которая присваивается.

При вычислении выражений тип каждого операнда может быть преобразован к другому типу. Преобразование типов может быть неявным при выполнении операций и в вызовах функций или явно определенным с помощью операций приведения типов.

3.1. ОПЕРАНДЫ

Операнд — это константа, идентификатор, строковый литерал, вызов функции, индексное выражение, выражение выбора элемента или более сложное выражение, сформированное комбинацией операндов и знаков операций или заключением операндов в скобки. Любой операнд, который имеет константное значение, называется *константным выражением*.

Каждый операнд имеет тип.

Константы

Если в качестве операнда используется константа, то ему соответствует значение и тип представляющей его константы:

- целая константа может быть одного из следующих типов: `int`, `long`, `unsigned int`, `unsigned long`, в зависимости от размера целого и от того, как задана его величина;
- символьная константа имеет тип `int`;
- константы с плавающей точкой всегда имеют тип `double`.

Строки

Строковый литерал состоит из последовательности символов, заключенных в кавычки. Он представляется в памяти как массив элементов типа `char` и инициализируется указанной последовательностью символов.

Значением строкового литерала является адрес первого элемента строки. Так как строковые литералы — это полноценные указатели, то они могут быть использованы в качестве операндов в выражениях, допускающих величины типа указателей, подчиняясь при этом тем же самым ограничениям. Однако так как строки не являются переменными, их нельзя использовать в левой части операции присваивания.

Последним символом строки всегда является нулевой символ, который автоматически добавляется при сохранении строки в памяти. Например, строка "три" содержит в действительности четыре символа, а не три

Идентификаторы

Идентификаторы именуют переменные и функции. Каждый идентификатор имеет тип, который устанавливается при его объявлении. Значение идентификатора зависит от типа следующим образом:

- идентификаторы объектов целых и плавающих типов представляют значения соответствующего типа;
- идентификатор объекта типа `enum` представляет значение одной константы из множества значений констант в перечислении. Значение идентификатора является константным значением. Тип значения есть `int`, что следует из определения перечисления;
- идентификатор объекта типа `struct` или `union` представляет значение, определенное структурой или смесью;
- идентификатор, объявленный как указатель, представляет указатель на значение заданного в объявлении типа;
- идентификатор, объявленный как массив, представляет указатель, значение которого является адресом первого элемента массива. Тип адресуемых указателем величин — это тип элементов массива. Например, если `massiv` объявлен как массив целых из 6 элементов, то идентификатор `massiv` представляет адрес массива, тогда как индексное выражение `massiv[5]` ссылается на целое значение, которое является шестым элементом массива. Адрес массива не изменяется во время выполнения программы, хотя значения отдельных элементов можно изменить. Значение указателя, представленное идентификатором массива, не является переменной, и поэтому идентификатор массива не может появляться в левой части операции присваивания;
- идентификатор, объявленный как функция, представляет указатель, значение которого является адресом функции, возвращающей значение определенного типа. Адрес функции не изменяется во время выполнения программы. Меняется только возвращаемое значение. Таким образом, идентификаторы функций не могут появляться в левой части операции присваивания.

Вызов функций

Вызов функции состоит из выражения, за которым следует необязательный список-выражений в круглых скобках:

выражение-1 ([список-выражений])

Значением *выражения-1* должен быть адрес функции (например, идентификатор функции). Значение каждого выражения из *списка-выражений* передается в функцию в качестве фактического аргумента.

Если операнд является вызовом функции, то он имеет тип и значение возвращаемого функцией значения.

Индексные выражения

Операндом выражения может быть элемент массива, определяемый с помощью индексного выражения. Формат:

выражение-1[*выражение-2*]

Квадратные скобки в формате индексного выражения являются элементом синтаксиса.

Тип индексного выражения является типом элементов массива, а значение представляет величину, адрес которой вычисляется суммированием значений выражения-1 и выражения-2. Обычно *выражение-1* — это любой указатель, например идентификатор массива, а *выражение-2* — это целочисленная величина. Однако требуется только, чтобы одно из выражений было указателем, а второе — целочисленной величиной. Поэтому выражение-1 может быть целой величиной, а выражение-2 — указателем. В любом случае выражение-2 должно быть заключено в квадратные скобки.

Индексное выражение обычно используется для ссылок на элементы массива, тем не менее индекс может появиться с любым указателем.

Индексное выражение для ссылки на одномерный массив вычисляется путем сложения целой величины со значением указателя с последующим применением к результату операции разадресации (*).

Так как одно из выражений, указанных в индексном выражении, является указателем, то при сложении используются правила *адресной арифметики*. Целочисленная величина сначала преобразуется к адресному представлению путем умножения ее на размер типа, адресуемого указателем.

Например, пусть идентификатор `row` был объявлен как массив элементов типа `float`:

```
float row[10];
```

Чтобы вычислить выражение `row[i]`, целая величина `i` умножается на размер типа `float`. Преобразованное значение `i` представляет смещение `i`-го элемента от начала массива `row`. Затем это значение складывается со значением указателя `row`, что дает адрес `i`-го элемента массива. И наконец, к полученному адресу применяется операция разадресации. Таким образом, результатом индексного выражения `row[i]` является значение `i`-го элемента массива `row`.

Примечание. Индексное выражение `row[0]` представляет значение первого элемента массива, так как отсчет смещения ведется от нуля. Следовательно, такое выражение, как `row[5]`, ссылается на шестой элемент массива.

Выражения с несколькими индексами ссылаются на элементы многомерных массивов. Многомерный массив — это массив, элементами которого являются массивы. Например, первым элементом трехмерного массива является массив с двумя измерениями.

Для ссылки на элемент многомерного массива индексное выражение должно иметь следующий формат:

выражение-1 [*выражение-2*][*выражение-3*]...

Данное индексное выражение интерпретируется слева направо. Сначала вычисляется самое левое индексное выражение:

выражение-1 [*выражение-2*]

Результат сложения выражения-1 и выражения-2 есть адресное выражение, с которым складывается выражение-3, и т.д. Операция разадресации осуществляется после вычисления последнего индексного выражения. Однако операция разадресации не применяется, если значение последнего указателя адресует величину типа массив.

Пусть имеется многомерный массив:

```
int array[2][5][2];
```

Выражение `array[1][2][2]` вычисляется следующим образом:

1) вычисляется выражение `array[1]`. Сначала индекс 1 умножается на размер элемента этого массива, т.е. размер типа `int`, умноженный на размер двумерного массива 5 на 2. Полученное значение складывается с указателем `array`. Результат будет указателем на второй двумерный массив (5*2) в трехмерном массиве `array`;

2) второй индекс 2 умножается на размер массива из двух элементов типа `int` и складывается с адресом, соответствующим `array[1]`;

3) так как каждый элемент двухэлементного массива — это величина типа `int`, то индекс 2 умножается на размер типа `int` перед сложением с адресом `array[1][2]`;

4) наконец, выполняется разадресация полученного указателя. Результатом выражения будет элемент типа `int`.

Если было указано выражение `array[1][2]`, то результатом будет указатель на массив из двух элементов типа `int`. Соответственно значением индексного выражения `array[1]` является указатель на двумерный массив.

Выражение выбора элемента

Выражение выбора элемента применяется, если в качестве операнда надо использовать элемент структуры или смеси. Такое выражение имеет значение и тип выбранного элемента.

Выражение выбора элемента имеет 2 формата.

Формат 1:

выражение.идентификатор

Формат 2:

выражение->идентификатор

В формате 1 *выражение* представляет величину типа `struct` или `union`, а *идентификатор* — это имя элемента структуры или смеси.

В формате 2 *выражение* представляет указатель на структуру или смесь, а *идентификатор* — это имя элемента структуры или смеси.

Оба формата выражения выбора элемента дают одинаковый результат. Действительно, запись, включающая знак операции выбора `->`, является сокращенной версией записи с точкой для случая, когда выражению, стоящему перед точкой, предшествует операция разадресации (`*`), примененная к указателю, т.е. запись

выражение-`>`идентификатор

эквивалентна

(`*выражение`).идентификатор

в случае, если выражение является указателем.

Пример.

```
struct tree {  
    float num;  
    int spisok[5];  
    struct tree *left;  
} tr[5], elem;  
elem.left=&elem;
```

В примере используется операция выбора `"."` для доступа к элементу `left` структурной переменной `elem`. Таким образом элементу `left` структурной переменной `elem` присваивается адрес самой переменной `elem`, т.е. переменная `elem` хранит ссылку на саму себя.

Выражение приведения типов

Приведение типов является методом явного преобразования типов объектов. Выражения приведения типов имеют следующий формат:

(*имя-типа*) операнд

Приведение типов используется для преобразования объектов одного скалярного типа в любой другой скалярный тип. Однако выражению с приведением типа не может быть присвоено другое значение.

Пример.

```
int    a;  
float  b;  
b = (float)a+2.0;
```

Целая переменная `"a"` с помощью операции приведения типов приводится к плавающему типу, а затем уже участвует в вычислении выражения.

Константное выражение

Константное выражение — это выражение, результатом которого является константа. Операндами константного выражения могут быть целые константы, символьные константы, константы с плавающей точкой, константы перечисления, выражения приведения типов, выражения с операцией `sizeof` и другие константные выражения. На использование знаков операций в константных выражениях налагаются следующие ограничения:

1) в константных выражениях нельзя использовать операции присваивания и последовательного вычисления `(,)`;

2) операция "адрес" `(&)` может быть использована только при некоторых инициализациях.

Выражения со знаками операций

В качестве операндов в выражениях могут участвовать выражения, включающие знаки операций. Выражения со знаками операций могут быть унарными, бинарными или тернарными.

Унарное выражение состоит из операнда с предшествующим знаком унарной операции и имеет формат:

уноп операнд

В формате унарного выражения *уноп* означает знак унарной операции; *операнд* может быть именем переменной или выражением приведения типа. В последнем случае выражение должно быть заключено в круглые скобки.

Бинарное выражение состоит из двух операндов, разделенных знаком бинарной операции, и имеет формат:

операнд биноп операнд

В формате бинарного выражения *биноп* — знак бинарной операции.

Тернарное выражение состоит из трех операндов, разделенных знаком тернарной операции `(?:)`, и имеет формат:

операнд ? операнд : операнд

Выражения со знаками операций включают в себя также выражения присваивания. В выражениях присваивания могут использоваться знаки унарных и бинарных операций присваивания.

Любой операнд может быть заключен в скобки. Они не влияют на тип и значение выражения, заключенного в скобки. Например, в выражении

`(20 - 15) / (15 - 10)`

скобки, заключающие выражения, означают, что величина (20 - 15) является левым операндом операции деления, а величина (15 - 10) — правым операндом. Результат выражения (20 - 15) / (15 - 10) равен 1. Без скобок значение записи 20 - 15 / 15 - 10 равнялось бы 9. Хотя скобки влияют на то, каким путем группируются операнды в выражении, они не определяют, в какой последовательности вычисляются операнды.

3.2. ОПЕРАЦИИ

Классификация операций

По количеству операндов, участвующих в операции, операции подразделяются на:

- унарные операции, или операции с одним операндом,
- бинарные операции, или операции с двумя операндами,
- тернарную операцию, или операцию с тремя операндами (это условная операция ?:).

Операции присваивания также могут быть унарными и бинарными.

Унарные операции перечислены по группам в табл. 3.1. Бинарные операции перечислены по группам в табл. 3.2.

Интерпретация унарных операций производится справа налево. Бинарные операции интерпретируются слева направо.

Таблица 3.1

Группы унарных операций

Знак операции	Операция	Группа операций
-	Арифметическое отрицание	Отрицание и дополнение
~	Поразрядное логическое отрицание (дополнение)	
!	Логическое отрицание	
*	Разадресация	Разадресация и адрес
&	Адрес	
+	Унарный плюс	Унарный плюс
sizeof	Размер	Размер

Таблица 3.2

Группы бинарных операций

Знак операции	Операция	Группа операций
*	Умножение	Мультипликативные
/	Деление	
%	Остаток от деления	
+	Сложение	Аддитивные
-	Вычитание	
<<	Сдвиг влево	Операции сдвига
>>	Сдвиг вправо	
<	Меньше	Операции отношения
>	Больше	
<=	Меньше или равно	
>=	Больше или равно	
=	Равно	
!=	Не равно	Поразрядные
&	Поразрядное логическое И	
	Поразрядное логическое ИЛИ	
↑	Поразрядное исключающее ИЛИ	
&&	Логическое И	Логические
	Логическое ИЛИ	
,	Запятая (последовательное вычисление)	Последовательного вычисления

Ниже приведены операции присваивания. Операция присваивания означает, что значение правого операнда должно быть присвоено области памяти с именем, заданным левым операндом. Поэтому левый операнд операции присваивания (или единственный операнд унарного выражения присваивания) должен быть выражением, ссылающимся на область памяти.

Операции присваивания

Знак операции	Название операции
++	Увеличение (унарный инкремент)
--	Уменьшение (унарный декремент)
=	Простое присваивание
*=	Умножение с присваиванием
/=	Деление с присваиванием
%=	Остаток от деления с присваиванием
+=	Сложение с присваиванием
-=	Вычитание с присваиванием
<<=	Сдвиг влево с присваиванием
>>=	Сдвиг вправо с присваиванием
&=	Поразрядное И с присваиванием
=	Поразрядное ИЛИ с присваиванием
^=	Поразрядное исключающее ИЛИ с присваиванием

Выражение, которое ссылается на область памяти, называется *именуемым выражением*, или *l-значением*.

Именуемыми выражениями могут быть следующие выражения:

идентификаторы данных целого и плавающего типов, типов указателя, структуры и смеси;

индексные выражения (`[ ]`), исключая индексные выражения, которые вычисляются как массив или функция;

выражения выбора элемента (`->` или `.`), если выбираемый элемент является одним из вышеперечисленных выражений;

выражения унарной операции разадресации (`*`), за исключением выражений, ссылающихся на массив или функцию;

именующие выражения (*l-значения*) в круглых скобках.

Примечание. Объект, объявленный с ключевым словом `const`, не может быть левым операндом операции присваивания.

Обычные арифметические преобразования

При выполнении операций производится автоматическое преобразование типов, чтобы привести операнды выражений к общему типу или чтобы расширить короткие величины до размера целых величин, используемых в машинных операциях. Выполняемые преобразования зависят от специфики операций и от типа операнда или операндов. Тем не менее для многих операций выполняются похожие преобразования целых и плавающих типов. Эти преобразования известны как арифметические преобразования, поскольку они применяются к типам величин, обычно используемым в арифметике.

Арифметические преобразования, приведенные ниже, называются *обычными арифметическими преобразованиями*.

Обычные арифметические преобразования осуществляются по следующим правилам:

1. Операнды типа `float` преобразуются к типу `double`.
2. Если один операнд типа `long double`, то второй операнд преобразуется к типу `long double`.
3. Если один операнд типа `double`, то второй операнд преобразуется к типу `double`.
4. Любые операнды типа `char` или `short` преобразуются к типу `int`.
5. Любые операнды типов `unsigned char` или `unsigned short` преобразуются к типу `unsigned int`.
6. Если один операнд типа `unsigned long`, то второй операнд преобразуется к типу `unsigned long`.
7. Если один операнд типа `long`, то второй операнд преобразуется к типу `long`.
8. Если один операнд типа `unsigned int`, то второй операнд преобразуется к типу `unsigned int`.

Таким образом, при вычислении выражения операнды преобразуются к типу того операнда, который имеет наибольший размер.

Пусть, например, объявлены следующие переменные:

```
double fl;  
unsigned char ch;  
unsigned long m;  
int i;
```

Тогда при вычислении выражения

```
fl * (i + ch/m)
```

преобразования будут выполняться следующим образом. Операнд `ch` сначала преобразуется к типу `unsigned int` (правило 5). Затем он преобразуется к `unsigned long` (правило 6). По этому же правилу `i` преобразуется к `unsigned long` и результат выражения, заключенного в круглые скобки, будет иметь тип `unsigned long`. Затем он преобразуется к `double` (правило 3) и результат всего выражения будет типа `double`.

Операции отрицания и дополнения

Операция арифметического отрицания ("−") вырабатывает отрицание своего операнда. Операнд должен быть целой или плавающей величиной. При выполнении осуществляются обычные арифметические преобразования.

Пример.

```
int stat = 5;  
stat = -stat; /* Значение stat является отрицанием 5 */  
stat = -12; /* stat присваивается отрицание к 12 */
```

Операция логического отрицания НЕ ("!") вырабатывает значение 0, если операнд есть истина (не ноль), и значение 1, если операнд есть ложь (0). Результат имеет тип `int`. Операнд должен быть целого или плавающего типа или типа указатель.

Пример.

```
int x = 0;  
if (!x);
```

Результат выражения `!x` равен 1 (истина).

Операция двоичного дополнения ("~") вырабатывает двоичное дополнение своего операнда. Операнд должен быть целого типа. Осуществляются обычные арифметические преобразования, результат имеет тип операнда после преобразования.

Пример.

```
main ()  
{  
char c = '9';  
unsigned char cc;  
cc = ~c;
```



```
printf ("с(символ) =    %c\tс(шестнадцатеричное)    =
                                %x\n",с,с);
printf ("~с(символ) =  %c\t~с(шестнадцатеричное)  =
                                %x\n",сс,сс);
}
```

Результат выполнения программы:

```
с(символ) = 9          с(шестнадцатеричное) = 39
~с(символ) = Ц        ~с(шестнадцатеричное) = с6
```

Операции разадресации и адреса

Эти операции используются для работы с переменными типа указатель.

Операция разадресации ("*") осуществляет косвенный доступ к адресуемой величине через указатель. Операнд должен быть указателем. Результатом операции является величина, на которую указывает операнд. Типом результата является тип величины, адресуемой указателем. Результат не определен, если указатель содержит недопустимый адрес.

Некоторыми общими ситуациями, когда указатель содержит недопустимый адрес, являются следующие:

- указатель является нулевым;
- указатель определяет адрес локального объекта, который не является активным в момент ссылки;
- указатель определяет адрес, который не выровнен должным образом для типа объекта, на который указывает указатель;
- указатель определяет адрес, не используемый выполняющейся программой.

Операция "адрес"("&") дает адрес своего операнда. Операндом может быть любое именуемое выражение. Имя функции или массива также может быть операндом операции "адрес", хотя в этом случае знак операции является излишним, так как имена функций и массивов являются адресами. Результатом операции "адрес" является указатель на операнд. Тип, адресуемый указателем, является типом операнда.

Операция "адрес" не может применяться к элементам структуры, являющимся полями битов, и к объектам с классом памяти register.

Пример.

```
int x, y=0, *address;
address = &x ; /* Переменной address присваиваем адрес
                переменной x */
*address = y; /* Переменной x присваиваем значение y */
```


Операция sizeof

С помощью операции sizeof можно определить размер памяти, которая соответствует идентификатору или типу. Выражение с операцией sizeof имеет следующий формат:

`sizeof выражение`

Выражение — это либо идентификатор, либо имя типа, заключенное в круглые скобки. Имя типа не может быть void. Идентификатор не может относиться к полю битов или быть именем функции.

Эта операция позволяет избежать использования в программах данных, размеры которых машинно-зависимы.

Если в качестве выражения указано имя массива, то результатом является размер всего массива, а не размер указателя, соответствующего идентификатору массива.

Когда операция sizeof применяется к имени типа структуры или смеси или к идентификатору, имеющему тип структуры или смеси, то результатом является фактический размер структуры или смеси, который может включать участки памяти, используемые для выравнивания элементов структуры или смеси. Таким образом, этот результат может не соответствовать размеру, вычисленному путем сложения размеров элементов структуры.

Пример.

```
main ()
{ struct str {
    char a;
    int b;
    double f;
} str;
printf("размер переменной str = %d\n", sizeof(str));
}
```

Результат выполнения программы:

размер переменной str = 12

Мультипликативные операции

К мультипликативным операциям относятся операции умножения ("*"), деления ("/") и получения остатка от деления ("%"). Операндами операции "%" должны быть целые числа. Операции умножения "*" и деления "/" выполняются над целыми и плавающими операндами. Типы первого и второго операндов могут отличаться. Для мультипликативных операций выполняются обычные арифметические преобразования операндов. Типом результата является тип операндов после преобразования.

Примечание. Преобразования, выполняемые мультипликативными операциями, не обеспечивают обработку ситуаций переполнения или потери значимости и могут привести к потере информации, если результат операции не может быть представлен в типе операндов после преобразования.

Операция умножения ("*") вызывает умножение операндов.

Примеры.

```
int a = 5;  
float y = 0.2;  
double x, z;  
x = y * a;
```

/* Значения переменных a и y преобразуются к типу double, а затем результат умножения присваивается переменной x */

```
z = (x - a) * y;
```

/* Значения переменных a и y перед вычислением выражения преобразуются к типу double */

Операция деления ("/") вызывает деление первого операнда на второй. Если две целые величины не делятся нацело, результат усекается по следующим правилам:

- если операнды положительны или типа `unsigned`, результат усекается к ближайшему меньшему целому;
- если один из операндов отрицателен, направление усечения результата (к нулю или от нуля) определяется реализацией (в данной реализации к нулю).

При попытке деления на нуль выдается сообщение времени выполнения.

Пример.

```
main ()  
{  
  int c = 49;  
  int cc = 10;  
  printf ("%d/%d = %d \n",c,cc,c/cc);  
  printf ("%d/-%d = %d \n",c,cc,c/(-cc));  
}
```

Результат выполнения программы:

```
49/10 = 4  
49/-10 = -4
```

Операция "остаток от деления" ("%") имеет результатом остаток от деления первого операнда на второй.

Знак результата зависит от конкретной реализации. В данной реализации знак результата совпадает со знаком первого операнда. Если второй операнд равен нулю, выдается сообщение.

Пример.

```
main ()  
{  
  int c = 49;  
  int cc = 10;
```



```
printf ("остаток от деления %d на %d = %d \n",c,cc,c%cc);
printf ("остаток от деления %d на -%d = %d \n",c,cc,c%(-cc));
printf ("остаток от деления -%d на %d = %d \n",c,cc,(-c)%cc);
printf ("остаток от деления -%d на -%d = %d \n",c,cc,(-c)%(-cc));
}
```

Результат выполнения программы:

```
остаток от деления 49 на 10 = 9
остаток от деления 49 на -10 = 9
остаток от деления -49 на 10 = -9
остаток от деления -49 на -10 = -9
```

Аддитивные операции

К аддитивным операциям относятся сложение ("+") и вычитание ("-"). Операнды могут быть целого или плавающего типов. В некоторых случаях операндами аддитивных операций могут быть указатели.

При аддитивных операциях выполняются обычные арифметические преобразования. Преобразования, выполняемые аддитивными операциями, не обеспечивают обработку ситуаций переполнения и потери значимости. Информация теряется, если результат аддитивной операции не может быть представлен типом операндов после преобразования. При этом не выдается никакого сообщения об ошибке.

Пример.

```
main ()
{
  int a = 30000;
  int b = 30000;
  int c;
  c = a + b;
  printf ("a = %d\t b = %d\t a + b = %d\n", a, b, c);
}
```

Результат выполнения программы:

```
a = 30000      b = 30000      a + b = -5536
```

Результатом выполнения операции сложения ("+") является сумма двух операндов. Операнды могут быть целого или плавающего типов или один операнд может быть указателем, а второй — целой величиной.

Когда целая величина складывается с указателем, то целая величина (i) преобразуется путем умножения ее на размер памяти, занимаемой величиной, адресуемой указателем. Когда преобразованная целая величина складывается с величиной указателя, то результатом является указатель, адресуемый ячейку памяти, расположенную на "i" позиций дальше от исходного адреса. Новое значение указателя адресует тот же самый тип данных, что и исходный указатель.

Операция вычитания (" - ") вычитает второй операнд из первого. Возможна следующая комбинация операндов:

- 1) оба операнда целого или плавающего типов;
- 2) оба операнда являются указателями на один и тот же тип;
- 3) первый операнд является указателем, а второй — целым.

Когда один указатель вычитается из другого, разность преобразуется к целой величине со знаком путем деления ее на длину типа, который определяется указателями. Результат представляет число позиций памяти данного типа между двумя адресами.

Когда целая величина вычитается из указателя, то перед выполнением операции производятся те же самые преобразования, что и при сложении целого с указателем. Результатом вычитания является указатель, адресуемый позицию памяти, расположенную на "i" позиций перед исходным адресом, где "i" — целое, а каждая позиция — это длина типа, адресуемого указателем. Новый указатель адресует тот же самый тип данных, что и исходный указатель.

Аддитивные операции, имеющие операндом указатель, являются операциями адресной арифметики. Операции адресной арифметики, применяемые к указателю и целому, имеют осмысленный результат только тогда, когда указатель адресует массив, а целая величина представляет смещение адреса в пределах этого массива. Когда целая величина преобразуется к адресному смещению, предполагается, что в пределах смещения расположены элементы одинакового размера.

Это предположение справедливо для элементов массива. По определению, массив — это последовательность величин одного и того же типа; его элементы расположены в смежных ячейках памяти. Способ запоминания для других типов не гарантирует непрерывного заполнения памяти. Сложение и вычитание адресов, ссылающихся на любые другие величины, кроме элементов массива, дает непредсказуемый результат.

Аналогично, когда вычитается один указатель из другого, предполагается, что указатели ссылаются на величины одного и того же типа и что нет неиспользованных участков памяти между элементами, расположенными в промежутке между адресами, заданными операндами.

Пример.

```
double mas[10], *prt;
```

```
int number;
```

```
prt = mas + 3; /* prt - указывает на четвертый элемент массива */
```

```
number = prt - mas; /* number = 3 */
```

Операции сдвига

Операции сдвига сдвигают операнд влево ("<<") или вправо (">>") на число битов, заданное вторым операндом. Оба операнда должны быть целыми величинами. Выполняются обычные арифметические

преобразования. Тип результата — это тип левого операнда после преобразования. При сдвиге влево правые освобождающиеся биты устанавливаются в нуль. При сдвиге вправо метод заполнения освобождающихся левых битов зависит от типа, полученного после преобразования первого операнда. Если тип `unsigned`, то свободные левые биты устанавливаются в нуль.

В противном случае они заполняются копией знакового бита. Результат операции сдвига не определен, если второй операнд отрицательный.

Преобразования, выполняемые операциями сдвига, не обеспечивают обработку ситуаций переполнения и потери значимости. Информация теряется, если результат операции сдвига не может быть представлен типом первого операнда после преобразования.

Примеры.

```
int value = 0x1234, rez1;
rez1 = value << 8;          /* rez1 = 0x3400 */
value = rez1 >> 8;          /* value = 0x34 */
```

Операции отношения

Операции отношения сравнивают первый операнд со вторым. Результатом операции отношения является 1, если проверяемое отношение истинно, и 0, если ложно. Тип результата `int`. Ниже приведены проверяемые отношения и знаки операций отношения языка Си.

Знак операции	Проверяемое отношение
<	Первый операнд меньше, чем второй операнд
>	Первый операнд больше, чем второй операнд
<=	Первый операнд меньше или равен второму операнду
>=	Первый операнд больше или равен второму операнду
==	Первый операнд равен второму операнду
!=	Первый операнд не равен второму операнду

Операнды могут быть целого, плавающего типов и типа указателя. Типы первого и второго операндов могут различаться. Над целыми и плавающими операндами выполняются обычные арифметические преобразования.

Кроме того, возможны следующие комбинации операндов в операциях отношения:

1) оба операнда любой операции отношения могут быть указателями на один и тот же тип. Для операции равенства (`==`) и неравенства (`!=`) результат сравнения показывает, адресуют ли оба указателя одну и ту же ячейку памяти или нет. Результат сравнения указателей для других операций (`<`, `>`, `<=`, `>=`) отражает относительное положение двух адресов памяти. Так как адрес данной величины произволен, то сравнение адресов двух несвязанных величин, вообще говоря, не име-

ет смысла. Однако сравнение адресов различных элементов одного и того же массива может быть полезным, так как элементы массива хранятся в последовательном порядке. Адрес первого элемента массива меньше, чем адрес следующего элемента;

2) указатель может быть проверен на равенство ($=$) или неравенство ($\neq$) константе 0. Указатель, имеющий значение 0, называется нулевым указателем и не указывает на ячейку памяти.

Пример.

```
#define MAX 100
```

```
...
for (i=0; i<=MAX; i++)
{ if (i % 5 == 0)
    printf ("%d\t", i)
};
```

Программа печатает все числа, кратные 5, в промежутке от 0 до 100.

Поразрядные операции

К поразрядным операциям относятся:

- операция поразрядного логического И ($\&$);
- операция поразрядного логического ИЛИ ($\mid$);
- операция поразрядного исключающего ИЛИ ($\wedge$).

Операнды поразрядных операций могут быть любого целого типа. Выполняются обычные арифметические преобразования; тип результата — это тип операндов после преобразования.

Операция поразрядного логического И (" $\&$ ") сравнивает каждый бит первого операнда с соответствующим битом второго операнда. Если оба сравниваемых бита единицы, то соответствующий бит результата устанавливается в 1, в противном случае — в 0.

Операция поразрядного логического ИЛИ (" $\mid$ ") сравнивает каждый бит первого операнда с соответствующим битом второго операнда. Если любой (или оба) из сравниваемых битов равен 1, то соответствующий бит результата устанавливается в 1, в противном случае — в 0.

Операция поразрядного исключающего ИЛИ (" $\wedge$ ") сравнивает каждый бит первого операнда с соответствующим битом второго операнда. Если один из сравниваемых битов равен 0, а второй бит равен 1, то соответствующий бит результата устанавливается в 1, в противном случае (оба бита равны 1 или 0) бит результата устанавливается в 0.

Пример.

```
int i = 0x45ff, y = 0xff;
short result;
result = i ^ y;      /* result 0x4500 */
result = i | y;      /* result 0x45ff */
result = i & y;      /* result 0xff */
```


Логические операции

К логическим операциям относятся:

- операция логического И (&&);
- операция логического ИЛИ (||).

Операнды логических операций могут быть целых типов, плавающих типов или типа указателя, при этом в каждой операции могут участвовать операнды различных типов.

Операнды логических выражений вычисляются слева направо. Если значения первого операнда достаточно, чтобы определить результат операции, то второй операнд не вычисляется.

Логические операции не выполняют стандартные арифметические преобразования. Они оценивают каждый операнд с точки зрения его эквивалентности нулю. Результатом логической операции является 0 или 1. Тип результата `int`.

Операция логического И ("&&") вырабатывает значение 1, если оба операнда имеют ненулевое значение. Если один из операндов равен 0, то результат также равен нулю. Если значение первого операнда равно нулю, то второй операнд не вычисляется.

Операция логического ИЛИ ("||") выполняет над операндами операцию включающего ИЛИ. Она вырабатывает значение 0, если оба операнда имеют значение 0; если какой-либо из операндов имеет ненулевое значение, то результат операции равен 1. Если первый операнд имеет ненулевое значение, то второй операнд не вычисляется.

Пример.

```
unsigned char tolower(unsigned char a)
{
    if(((a >= 'A') && (a <= 'Z')) || ((a >= 'a') && (a <= 'z')))
        return(a+0x20);
    if (a == 'Ё')
        return('ё');
    return(a);
}
```

Описанная в примере функция `tolower` переводит символ, если он является прописной буквой, в строчную букву. Для проверки, является ли символ прописной буквой, используются логические операции.

Операция последовательного вычисления

Операция последовательного вычисления, обозначаемая запятой (,), используется обычно для вычисления двух или более выражений там, где по синтаксису допустимо только одно выражение. Эта операция вычисляет два операнда последовательно слева направо. При выполнении операции последовательного вычисления преобразование типов не производится. Операнды могут быть любых типов. Результат

операции имеет значение и тип второго операнда. Так как запятая может использоваться как разделитель, следует различать по контексту, является ли запятая знаком операции или разделителем.

```
/****** Пример 1 *****/  
for (i=0,y=0; i<MAX && y<MAX; i++)
```

В этом примере запятая используется как операция последовательного вычисления.

```
/****** Пример 2 *****/  
funct (arg1, arg2, arg3)
```

В этом примере запятые используются в качестве разделителей.

Условная операция

В языке Си есть одна тернарная операция: условная операция, которая имеет следующий формат:

операнд-1 ? операнд-2 : операнд-3

Операнд-1 должен быть целого или плавающего типа или типа указатель. Он оценивается с точки зрения его эквивалентности 0.

Если операнд-1 не равен нулю, то вычисляется *операнд-2*, и его значение является результатом операции. Если операнд-1 равен 0, то вычисляется *операнд-3*, и его значение является результатом операции.

Заметим, что вычисляется либо операнд-2, либо операнд-3, но не оба.

Тип результата зависит от типов операнда-2 и операнда-3 следующим образом:

1) если операнд-2 или операнд-3 имеет целый или плавающий тип (их типы могут отличаться), то выполняются обычные арифметические преобразования. Типом результата является тип операнда после преобразования;

2) если операнд-2 и операнд-3 имеют один и тот же тип структуры, смеси или указателя, то тип результата будет тем же самым типом структуры, смеси или указателя;

3) если оба операнда имеют тип `void`, то результат имеет тип `void`;

4) если один операнд является указателем на объект любого типа, а другой операнд является указателем на `void`, то указатель на объект преобразуется к указателю на `void`, который и будет типом результата;

5) если один из операндов является указателем, а другой — константным выражением со значением 0, то типом результата будет тип указателя.

Пример.

```
max = (a <= b) ? b : a;
```

В этом примере переменной `max` присваивается максимальное значение переменных `a` и `b`.

Операции увеличения и уменьшения

Операции увеличения (++) и уменьшения (--) являются унарными операциями присваивания. Они соответственно увеличивают или уменьшают значение операнда на единицу. Операнд должен быть целого или плавающего типа или типа указатель и быть модифицируемым именованным выражением (без слова `const`). Операнд целого или плавающего типа увеличивается или уменьшается на 1. Тип результата соответствует типу операнда. Операнд адресного типа увеличивается или уменьшается на размер объекта, который он адресует.

В языке допускается префиксная и постфиксная формы операций уменьшения и увеличения. Значение выражения, использующего операции увеличения и уменьшения, зависит от того, какая форма операций используется.

Если знак операции стоит перед операндом (префиксная форма), результатом операции является увеличенное или уменьшенное значение операнда.

Если знак операции стоит после операнда (постфиксная форма), значением выражения будет значение операнда. После использования этого результата значение операнда увеличивается или уменьшается.

Пример.

```
main()
{
    int x = 0, y = 0;
    int z, w;
    printf ("Начальные значения: x = %d\t y = %d\n", x, y);
    z = x++;
    w = ++y;
    printf ("\nПосле вычислений: x = %d\t z = %d\n", x, z);
    printf ("\t\t y = %d\t w = %d\n", y, w);
```

Результат выполнения программы:

```
Начальные значения:  x = 0    y = 0
После вычислений:   x = 1    z = 0
                    y = 1    w = 1
```

Простое присваивание

Операция простого присваивания используется для замены значения левого операнда значением правого операнда. При присваивании производятся преобразования типов по правилам, изложенным в п. 3.5.

```
/****** Пример 1 *****/
int x;
char y;
long z;
x = y;
```

Значение переменной `y` преобразуется к `int` и присваивается переменной `x`.

/****** Пример 2 *****/

$x = y + z;$

Значение переменной y преобразуется к типу `long`; вычисляется сумма $(y+z)$, и результат преобразуется к типу `int`, а затем присваивается переменной x .

Составное присваивание

Кроме простого присваивания, имеется целая группа операций присваивания, которые объединяют простое присваивание с одной из бинарных операций. Такие операции называются составными операциями присваивания и имеют следующий вид:

операнд1 биноп= операнд2

Это выражение трактуется следующим образом:

операнд1 = операнд1 биноп операнд2

т.е. при выполнении составного присваивания вначале выполняется бинарная операция над операндами, а затем результат присваивается операнду1. Однако выражение составного присваивания не эквивалентно расширенному формату, так как в составном присваивании операнд1 вычисляется только один раз, а в расширенном формате — дважды.

Операнды операции составного присваивания должны быть целого или плавающего типа. Каждая операция составного присваивания выполняет преобразования, которые осуществляются соответствующей бинарной операцией. Левым операндом операций сложения с присваиванием (" $+=$ ") и вычитания с присваиванием (" $-=$ ") может быть указатель, а правый операнд должен быть целым числом. Результатом операции составного присваивания являются значение и тип левого операнда.

Примеры.

`float n1 = 2.0;`

`double n2 = 5.3;`

`n1 *= n2; /* n1 = 2.0 * 5.3 = 10.6 */`

`n1 -= n2 - 0.3 /* n1 = 10.6 - (5.3 - 0.3) = 5.6 */`

3.3. ПРИОРИТЕТЫ ОПЕРАЦИЙ И ПОРЯДОК ВЫЧИСЛЕНИЯ

Приоритеты и порядок вычисления операций языка Си влияют на способы группирования и вычисления операндов в выражениях. Приоритет операции имеет смысл только при наличии нескольких операций с разными приоритетами. Выражения с наивысшим приоритетом вычисляются первыми.

Приоритеты операций и порядок выполнения операций приведены в табл. 3.3. Первый приоритет является наивысшим.

Из табл. 3.3 следует, что операнды, состоящие из константы, идентификатора, строки, вызова функции, индексного выражения, выражения выбора элемента или выражения в скобках, имеют наивысшие приоритеты и выполняются слева направо. Приведение типов имеет тот же приоритет и порядок выполнения, что и унарные операции.

Выражение может содержать несколько операций одного приоритета. Когда несколько таких операций появляются на одном и том же уровне в выражении, обработка производится в соответствии с порядком выполнения либо справа налево, либо слева направо. Направление выполнения не влияет на результат выражения, которое включает несколько операций умножения ("*"), сложения ("+") или поразрядных операций("&", "|", "^") на одном и том же уровне. Компилятор может вычислять такие выражения в любом порядке, даже в случае, когда в выражении появляются скобки, задающие порядок вычисления.

Таблица 3.3

Приоритеты и порядок выполнения операций

Приоритет	Знак операции	Тип операции	Порядок выполнения
1	() [] . ->	Выражение	Слева направо
2	- ~ ! * & ++ -- sizeof Приведение типов	Унарные	Справа налево
3	* / %	Мультипликативные	Слева направо
4	+ -	Аддитивные	Слева направо
5	<< >>	Сдвиг	Слева направо
6	< > <= >=	Отношение (неравенство)	Слева направо
7	== !=	Отношение (равенство)	Слева направо
8	&	Поразрядное И	Слева направо
9	^	Поразрядное исключающее ИЛИ	Слева направо
10		Поразрядное ИЛИ	Слева направо
11	&&	Логическое И	Слева направо
12		Логическое ИЛИ	Слева направо
13	?:	Условная	Справа налево
14	= *= /= %= += -= <<= >>=	Простое и составное присваивание	Справа налево
15	,	Последовательное вычисление	Слева направо

Некоторые операции гарантируют, что левый операнд будет обрабатываться первым. Эти операции устанавливают так называемые "точки последовательности". К ним относятся:

- операция логического И (&&);
- операция логического ИЛИ (||);
- тернарная операция (?:);
- операция последовательного вычисления (,);
- операция вызова функции.

Точками последовательности также являются конец полного выражения, любой инициатор, выражение в операторе выражения, условные выражения в операторах выбора (if либо switch) и цикла (do, while или for) и выражение в операторе return.

Операция последовательного вычисления обеспечивает вычисление операндов слева направо.

Логические операции также обеспечивают вычисление операндов слева направо. Однако логические операции вычисляют минимальное число операндов, необходимое для определения результата выражения. Таким образом, некоторые операнды выражения могут быть не вычислены.

Например, если в выражении $(x-5) \&\& ++i$ $x-5$ равно нулю (ложь), то $++i$ не будет вычисляться.

/***** Пример 1 *****/

$y = x/2 + a*5\%10;$

Порядок вычисления:

$y = (x/2) + ((a*5)\%10);$

/***** Пример 2 *****/

$a = b+++c/5;$

Порядок вычисления:

$a = (b++)+(c/5);$

3.4. ПОБОЧНЫЕ ЭФФЕКТЫ

Операции присваивания в сложных выражениях могут вызывать побочные эффекты, так как они изменяют значение переменной. Побочный эффект может возникнуть и при вызове функции, если он содержит прямое или косвенное присваивание (через указатель). Это связано с тем, что аргументы функции могут вычисляться в любом порядке. Например, побочный эффект имеет место в следующем вызове функции:

compute (a, a=k*2);

В зависимости от того, какой аргумент вычисляется первым, в функцию будут переданы различные значения.

Порядок вычисления операндов некоторых операций зависит от реализации и поэтому могут возникнуть побочные эффекты, если в одном из операндов используются операции унарного увеличения или уменьшения или другие операции присваивания.

Например, следующее выражение

$i*j+(j++)+(-i)$

может принимать различные значения при обработке разными компиляторами. Чтобы избежать побочных эффектов, надо придерживаться следующих правил:

1. Не использовать операции присваивания переменной в вызове функции, если эта переменная участвует в формировании других аргументов функции.

2. Не использовать операции присваивания переменной в выражении, если эта переменная включена в выражение более одного раза.

3.5. ПРЕОБРАЗОВАНИЕ ТИПОВ

При выполнении операций происходит неявное преобразование типов в следующих случаях:

- при выполнении операции осуществляются обычные арифметические преобразования (были рассмотрены в п. 3.2);
- при выполнении присваивания, если значение одного типа присваивается переменной другого типа;
- при передаче аргументов функции.

Кроме того, есть возможность явного приведения значения одного типа к другому.

В операциях присваивания тип значения, которое присваивается, преобразуется к типу переменной, получающей это значение. Допускается преобразование целых и плавающих типов, даже если такое преобразование ведет к потере информации.

Преобразование целых типов со знаком

Целое со знаком преобразуется к более короткому целому со знаком посредством усечения старших битов. Целое со знаком преобразуется к более длинному целому со знаком путем размножения знака.

При преобразовании целого со знаком к целому без знака целое со знаком преобразуется к размеру целого без знака и результат интерпретируется как значение без знака.

Преобразование целого со знаком к плавающему типу происходит без потери информации, за исключением случая преобразования значения типа `long int` или `unsigned long int` к типу `float`, когда точность частично может быть потеряна.

Преобразование целых типов без знака

Целое без знака преобразуется к более короткому целому без знака или со знаком путем усечения старших битов. Целое без знака преобразуется к более длинному целому без знака или со знаком путем расширения нулем.

Когда целое без знака преобразуется к целому со знаком того же размера, битовый шаблон не изменяется. Однако значение, которое оно представляет, изменяется, если знаковый бит установлен.

Целые значения без знака преобразуются к плавающему типу путем преобразования целого без знака к значению типа `signed long`, а затем значения `signed long` — в плавающий тип. Преобразования из `unsigned long` к типу `float`, `double` или `long double` производятся с потерей информации, если преобразуемое значение больше, чем максимальное положительное значение, которое может быть представлено для типа `long`.

Преобразование плавающих типов

Величины типа `float` преобразуются к `double` без изменения значения. Величины `double`, преобразованные к `float`, представляются точно, если возможно. Если значение слишком велико для `float`, то точность теряется.

При преобразовании величин с плавающей точкой к целым типам они сначала преобразуются к типу `long` (дробная часть плавающей величины при этом отбрасывается), а затем величины типа `long` преобразуются к требуемому целому типу. Если значение слишком велико для `long`, то результат преобразования не определен.

Преобразования из `float`, `double` или `long double` к типу `unsigned long` производятся с потерей точности, если преобразуемое значение больше, чем максимально возможное положительное значение, представляемое типом `long`.

Преобразование типов указателя

Указатель на величину одного типа может быть преобразован к указателю на величину другого типа. Однако результат может быть неопределенным из-за отличий в требованиях к выравниванию и размерах для различных типов.

Указатель на тип `void` может быть преобразован к указателю на любой тип, и указатель на любой тип может быть преобразован к указателю на тип `void` без ограничений.

В СП Си специальные ключевые слова `near`, `far` и `huge` позволяют модифицировать размер указателей в программах. Например, 16-битовый указатель может быть преобразован в 32-битовый для обращения к сегменту.

Значение указателя может быть также преобразовано к целой ве-

личине. Метод преобразования зависит от размера указателя и размера целого типа следующим образом:

- если размер указателя больше размера целого типа или равен ему, то указатель преобразуется точно так же, как целое без знака, но не может быть преобразован к величине с плавающей точкой;
- если указатель меньше, чем размер целого типа, то указатель вначале преобразуется к указателю с тем же размером, что и целый тип, и затем преобразуется к целому типу.

И наоборот, целый тип может быть преобразован к адресному типу по следующим правилам:

- если целый тип того же размера, что и указатель, то целая величина просто рассматривается как указатель (как целое без знака);
- если размер целого типа отличен от размера указателя, то целый тип вначале преобразуется к размеру указателя (при этом используются методы преобразования, описанные выше), а затем полученное значение трактуется как указатель.

Преобразования при вызовах функций

Преобразования, выполняемые над аргументами в вызове функции, зависят от того, был ли задан прототип функции (объявление "вперед") со списком объявлений типов аргументов.

Если задан прототип функции и он включает объявления типов аргументов, то компилятор осуществляет контроль типов.

Если прототип функции не задан или он задан без списка типов аргументов, то над аргументами в вызове функции выполняются только обычные арифметические преобразования. Эти преобразования выполняются независимо для каждого аргумента. Величины типа `float` преобразуются к `double`, величины типов `char` и `short` преобразуются к `int`, величины типов `unsigned char` и `unsigned short` преобразуются к `unsigned int`.

Если используются специальные ключевые слова `near`, `far` и `huge`, то могут быть также выполнены неявные преобразования значений типа указатель, передаваемых функции. Задавая прототип функции, можно переопределить эти неявные преобразования и позволить компилятору выполнить контроль типов.

Преобразование типов в операции приведения типов

Явное преобразование типа может быть сделано посредством операции приведения типов, которая имеет формат:

(имя-типа) операнд

где *имя-типа* задает тип, а *операнд* является величиной, которая должна быть преобразована к заданному типу.

Имя типа `void` можно использовать в операции приведения, но объект типа `void` не может быть присвоен другому объекту.

Глава 4

ОПЕРАТОРЫ

Все операторы языка могут быть условно разделены на следующие категории:

- условные операторы, к которым относится оператор условия if и оператор выбора switch;
- операторы цикла (for, while и do);
- операторы переходов (break, continue, return и goto);
- другие операторы (оператор "выражение", пустой оператор).

Операторы в программе могут объединяться в составные операторы с помощью фигурных скобок. Любой оператор в программе может быть помечен меткой, состоящей из имени и следующего за ним двоеточия.

Все операторы языка Си, кроме составных операторов, заканчиваются точкой с запятой (;).

4.1. ОПЕРАТОР "ВЫРАЖЕНИЕ"

Любое выражение, которое заканчивается точкой с запятой, является оператором. Выполнение оператора "выражение" заключается в вычислении выражения. Правила вычисления выражений были описаны в предыдущей главе. Присваивания и вызовы функций также являются выражениями.

```
/****** Пример 1 *****/  
++i;
```

Этот оператор представляет выражение, которое увеличивает значение переменной i на единицу.

```
/****** Пример 2 *****/  
cosinus=cos(a*5);
```

Этот оператор представляет выражение, включающее операцию присваивания и вызов функции.

```
/****** Пример 3 *****/  
SWAP(&x,&y);
```

Этот оператор представляет выражение, состоящее из вызова функции.

4.2. ПУСТОЙ ОПЕРАТОР

Пустой оператор состоит только из точки с запятой. При выполнении этого оператора ничего не происходит. Он обычно используется в следующих случаях:

- в операторах `do`, `for`, `while`, `if` в случаях, когда тело оператора не требуется, хотя по синтаксису требуется хотя бы один оператор;
- при необходимости пометить фигурную скобку меткой.

Синтаксис языка Си требует, чтобы после метки обязательно следовал оператор. Фигурная же скобка оператором не является. Поэтому, если надо передать управление на фигурную скобку, необходимо использовать пустой оператор.

Пример.

```
main ()
{
    ...
    for (...) {
        for (...) {
            while (...) {
                if (...) goto end;
                ...
            }
        }
    }
    end ;;
}
```

4.3. СОСТАВНОЙ ОПЕРАТОР

Формат:

```
{  [объявление]
    ...
    оператор
    [оператор]
    ...
}
```

Заметим, что в конце составного оператора точка с запятой не ставится.

Все объявления, включенные в составной оператор, должны быть в начале. Обычно составной оператор является телом другого оператора. Выполнение составного оператора заключается в последовательном выполнении составляющих его операторов.

4.4. ОПЕРАТОР IF

Формат оператора:

```
if (выражение)
    оператор-1
[else оператор-2]
```

Выполнение оператора `if` начинается с вычисления *выражения*. Далее выполнение идет по следующей схеме:

- если выражение истинно (не нуль), то выполняется *оператор-1*;
- если выражение ложно, то выполняется *оператор-2*;
- если выражение ложно и отсутствует фраза *else*, то выполняется следующий за *if* оператор.

После выполнения оператора *if* управление передается на следующий оператор программы.

Пример.

```
if (i < j)
    i++;
else {
    j = i - 3;
    i++;
}
```

Допускается использование вложенных операторов *if*. Оператор *if* может быть вложен во фразу *if* или во фразу *else* другого оператора *if*.

Чтобы сделать программы более понятными, рекомендуется группировать операторы и фразы во вложенных операторах *if*, используя фигурные скобки. Если же фигурные скобки опущены, компилятор связывает каждое ключевое слово *else* с наиболее близким *if*, для которого нет *else*.

Пример.

```
main ()
{
    int a=2, b=7, c=3;
    if (a > b)
    {
        if (b < c)
            c = b;
    }
    else c = a;
    printf ("c = %d/n", c);
}
```

Результат выполнения программы:
c = 2

Если в этой программе опустить фигурные скобки в операторе *if*, то программа будет иметь следующий вид:

```
main ()
{
    int a=2, b=7, c=3;
    if (a > b)
        if (b < c)
            c = b;
    else c = a;
    printf ("c = %d/n", c);
}
```

В этом случае ключевое слово *else* относится ко второму оператору *if*.

Результат выполнения программы:
c = 3

Пример вложенных операторов if:

```
char sign;  
int x,y,z;  
...  
if (sign == '-')  
    x=y-z;  
else if (sign == '+')  
    x=y+z;  
else if (sign == '*')  
    x=y*z;  
else if (sign == '/')  
    x=y/z;  
else printf ("плохой знак операции \n");
```

На этом примере видно, что такие конструкции получаются длинными и не всегда понятными. Другим способом организации выбора из множества различных вариантов является использование специального оператора выбора switch.

4.5. ОПЕРАТОР SWITCH

Оператор switch предназначен для организации выбора из множества различных вариантов.

Формат:

```
switch (выражение) {  
    [объявление]  
    ...  
    [case константное-выражение:]  
    ...  
    [список-операторов]  
    ...  
    [default:  
        [список-операторов]  
    ]  
}
```

Выражение, следующее за ключевым словом switch в круглых скобках, может быть любым выражением, допустимым в языке Си, значение которого должно быть целым.

Значение этого выражения является критерием для выбора из нескольких вариантов. Тело оператора switch состоит из нескольких операторов, помеченных ключевым словом case с последующим *константным-выражением*.

Так как константное выражение вычисляется во время компиляции, оно не может содержать переменные или вызовы функций. Обычно в качестве константных выражений используются целые или символьные константы.

Все константные выражения в операторе switch должны быть уникальными. Кроме операторов, помеченных ключевым словом case, мо-

жет быть, но не обязательно, один фрагмент, помеченный ключевым словом `default`.

Список-операторов может быть пустым либо содержать один или более операторов. Причем в операторе `switch` не требуется заключать операторы в фигурные скобки.

Схема выполнения оператора `switch`:

- вычисляется выражение в круглых скобках;
- вычисленное значение последовательно сравнивается с константными выражениями, следующими за `case`;
- если сравнение найдено, управление передается на оператор, помеченный соответствующим оператором `case`;
- если сравнение не найдено и есть вариант `default`, то управление передается на оператор, помеченный ключевым словом `default`;
- если сравнение не найдено и варианта `default` нет, то управление передается на следующий за `switch` оператор.

Вариант `default` может быть не последним в теле оператора `switch`.

Метки `case` и `default` в теле оператора `switch` существенны только при начальной проверке, когда определяется начальная точка выполнения тела оператора `switch`. Все операторы между начальным оператором и концом тела выполняются вне зависимости от меток, если только какой-то из операторов не передаст управление из тела оператора `switch`. Таким образом, программист должен сам позаботиться о выходе из `case`, если необходимо. Чаще всего для этой цели используется оператор `break`.

Примечание. В заголовке составного оператора, образующего тело оператора `switch`, могут присутствовать объявления, но если в объявлениях присутствует инициализация, то она не будет выполнена.

Для того чтобы выполнить одни и те же действия для разных значений выражения, можно пометить один и тот же оператор несколькими фразами `case`.

```
/****** Пример 1 *****/
int a = 2;
switch (a) {
    case 1 : func1 ();
    case 2 : func2 ();
    case 0 :
    case 4 : func3 ();
    default: printf("good bye\n");
}
```

Выполнение этого оператора начинается с оператора, помеченного `case 2`. Таким образом, будут выполнены функции `func2`, `func3` и `printf`. Для значений 0 и 4 переменной "а" будут выполнены одни и те же действия, а именно, выполнены функции `func3` и `printf`.


```

    /***** Пример 2 *****/
char sign;
int x,y,z;
switch (sign) {
    case '-': x = y-z;
                break;
    case '+': x = y+z;
                break;
    case '*': x = y*z;
                break;
    case '/': x = y/z;
                break;
    default: printf("неизвестная операция \n");
}

```

Легко убедиться, что это пример из п. 4.4, переписанный с применением оператора switch.

Если оператор switch используется в качестве оператора в теле внешнего оператора switch, то в вариантах case внешнего и внутреннего switch могут использоваться одинаковые константы.

Пример.

```

switch (val1) {
    case 0 :
        switch (val2) {
            case 1 : funct1();
                        break;
            case 0 : printf("функция не выполняется \n");
        }
    case 3 :
        ...}

```

4.6. ОПЕРАТОР BREAK

Оператор break обеспечивает прекращение выполнения самого внутреннего из объемлющих его операторов switch, do, for и while. После выполнения оператора break управление передается оператору, следующему за прерванным.

4.7. ОПЕРАТОР FOR

Оператор цикла for имеет следующий формат:

```

for ([выражение-инициализации]; [условное-выражение];
    [выражение-итерации])
    оператор

```

Выражение-инициализации обычно используется для установки начального значения переменных, управляющих циклом. Условное-выражение — это выражение, определяющее условие, при котором оператор цикла будет выполняться. Выражение-итерации определяет

изменение переменных, управляющих циклом после каждого выполнения цикла.

Схема выполнения оператора `for`:

- 1) вычисляется выражение инициализации;
- 2) вычисляется условное выражение;
- 3) если значение условного выражения не равно нулю (истина), выполняется оператор;
- 4) вычисляется выражение итерации;
- 5) вновь вычисляется условное выражение;
- 6) как только условное выражение становится равным нулю, управление передается на оператор, следующий за оператором `for`.

Существенно то, что проверка условия всегда выполняется в начале цикла. Это значит, что цикл может ни разу не выполниться, если условное выражение сразу будет ложным.

П р и м е р простого использования оператора `for`:

```
main ()
{ int i;
  for (i=0; i < 10; i++)
    printf ("квадрат числа %d=%d\n", i, i*i);
}
```

В этой программе вычисляются квадраты чисел от 0 до 9.

Допускаются некоторые варианты оператора `for`, которые повышают его гибкость в определенных ситуациях. Одним из таких вариантов является использование нескольких переменных, управляющих циклом. Например:

```
main ()
{ int top, bot;
  char string[100], temp;
  for (top=0, bot=100; top < bot; top++, bot--);
  { temp=string[top];
    string[top]=string[bot];
    string[bot]=temp;
  }
}
```

В этой программе строка символов записывается в обратном порядке.

Другим вариантом использования оператора `for` является бесконечный цикл. Для организации бесконечного цикла можно использовать пустое условное выражение в заголовке оператора `for`.

П р и м е р простейшего бесконечного цикла:

```
for (;;) {операторы}
```

Для выхода из такого цикла обычно используется оператор `break`.

Так как по синтаксису языка Си оператор может быть пустым, тело оператора `for` также может быть пустым. Такой формат оператора может использоваться для организации задержки выполнения.

Например:

```
for (i=0; i < 100000; i++) ;
```

4.8. ОПЕРАТОР WHILE

Оператор цикла `while` имеет следующий формат:

```
while ( выражение )  
    оператор
```

Выражение может быть любым, допустимым в языке Си выражением. *Оператор* может быть либо пустым, либо простым, либо составным.

Схема выполнения оператора `while`:

- 1) вычисляется выражение;
- 2) если выражение ложно (равно нулю), то тело оператора `while` не выполняется, а управление передается на следующий за `while` оператор;
- 3) если выражение истинно (не нуль), то тело оператора `while` выполняется;
- 4) процесс повторяется с шага 1.

Оператор цикла

```
for ( выражение-1; выражение-2; выражение-3 )  
    оператор;
```

может быть представлен оператором `while` следующим образом:

```
выражение-1;  
while ( выражение-2 ) {  
    оператор;  
    выражение-3;  
}
```

Так же, как и в операторе `for`, при выполнении оператора `while` вначале вычисляется выражение. Поэтому оператор `while` удобно использовать в ситуациях, если тело оператора не всегда надо выполнять. Он заменяет необходимость отдельной проверки перед циклом.

Пример.

```
unsigned char str[ ] = "РЕЗУЛЬТАТ";  
void center( int );  
/* Программа центровки текста */  
main()  
{  
    int len = 0;  
    while (str[len++] != '\0'); /* Подсчет длины строки */  
    center(len);  
    printf(str);  
}
```



```
void center(int l)    /* Функция выдачи начальных пробелов */
{
    l = (80 - l) / 2;
    while (l-- > 0)
        printf(" ");
}
```

Результат выполнения программы:

РЕЗУЛЬТАТ

4.9. ОПЕРАТОР DO

Оператор цикла `do` используется в тех случаях, когда тело цикла должно выполняться хотя бы один раз.

Формат оператора `do`:

```
do
    оператор
    while (выражение);
```

Схема выполнения оператора `do`:

- 1) выполняется *оператор*;
- 2) вычисляется *выражение*. Если выражение не равно нулю (истинно), то выполнение продолжается с шага 1. Если выражение равно нулю (ложно), то управление передается следующему оператору программы.

Чтобы прервать цикл до того, как условие станет ложным, можно использовать оператор `break`.

Пример.

```
/* Функция выдачи подсказки */
unsigned char prompt (ques)
unsigned char *ques;    /* Указатель на вопрос */
{
    unsigned char ans, rest;
    printf ("%s?\n", ques);
    do {
        printf ("Д(да) или Н(нет):");
        rest=ans=getchar();
        while (rest != EOF && rest != '\n')
            rest=getchar();    /* Пропустить остаток строки */
    } while (ans != 'Д' && ans != 'Н' && ans != EOF);
    return (ans);
}
```

Функция `prompt` выдает вопрос и возможные ответы на него. До тех пор, пока не будет введен один из правильных ответов, функция будет выдавать подсказку. В данном примере используется оператор `do-while` и вложенный в него оператор `while`.

4.10. ОПЕРАТОР CONTINUE

Оператор `continue` работает подобно оператору `break`, но в отличие от оператора `break` прерывает выполнение тела цикла и передает управление на следующую итерацию.

Формат оператора `continue`:

`continue;`

Пример вывода четных чисел:

```
main ()
{ int number;
  for (number = 0; number < 100; number++)
    { if (number % 2) continue;
      printf ("%d ", number);
    }
}
```

Когда переменная `number` становится нечетной, выражение `(number % 2)` получает значение 1 и выполняется оператор `continue`, который передает управление на следующую итерацию цикла `for`, не выполняя функцию `printf`.

Оператор `continue` так же, как и оператор `break`, прерывает самый внутренний из объемлющих его циклов.

4.11. ОПЕРАТОР RETURN

Оператор `return` завершает выполнение функции, в которой он задан, и возвращает управление в вызывающую функцию. Управление передается в вызывающую функцию в точку, непосредственно следующую за вызовом.

Формат оператора `return`:

`return [выражение];`

Значение *выражения*, если оно задано, возвращается в вызывающую функцию в качестве значения вызываемой функции. Если выражение опущено, то возвращаемое функцией значение не определено. Выражение может быть заключено в круглые скобки.

Если оператор `return` отсутствует в вызываемой функции, то управление автоматически передается в вызывающую функцию после выполнения последнего оператора функции. Возвращаемое функцией значение в этом случае не определено. Если функция не возвращает значения, ее следует объявлять с типом `void`.

Итак, оператор `return` используется в двух случаях: если надо немедленно выйти из функции и если функция должна возвращать значение.


```
/****** Пример 1 *****/  
int sum (int a , int b)  
{ return (a+b);  
}
```

Функция `sum` возвращает сумму аргументов. Оператор `return` используется для возврата значения.

```
/****** Пример 2 *****/  
void print (char x)  
{ if (x == 0)  
  { printf ("Это плохой аргумент \n");  
    return;  
  }  
  printf ("Введен аргумент %c\n",x);  
}
```

В этом примере оператор `return` используется для выхода из функции в случае, если аргумент равен нулю.

4.12. ОПЕРАТОР GOTO

Хотя для написания структурированных программ не рекомендуется использовать операторы перехода, в языке Си имеется оператор перехода `goto`.

Формат оператора `goto`:

```
goto имя;  
...  
имя: оператор
```

Оператор `goto` передает управление на оператор с меткой *имя*. Помеченный *оператор* должен находиться в той же функции, что и оператор `goto`, и метка должна быть уникальна.

Если помеченный оператор встречается в любом другом контексте, он выполняется без учета метки.

Метка — это просто идентификатор. Каждая метка должна отличаться от других меток той же функции.

Любой оператор в составном операторе может иметь метку. Используя оператор `goto`, можно передать управление внутрь составного оператора. Однако надо быть осторожным, передавая управление внутрь составного оператора, так как составной оператор может содержать объявления переменных с инициализацией. Так как обычно объявления располагаются в составном операторе до выполнимых операторов, то при передаче управления внутрь составного оператора на выполнимый оператор будет обойдена инициализация. Результат в этом случае непредсказуем.

Глава 5

ФУНКЦИИ

5.1. ОСНОВНЫЕ ПОНЯТИЯ

Мощность языка Си во многом объясняется легкостью и гибкостью в определении и использовании функций в Си-программах. В отличие от других языков программирования высокого уровня в языке Си нет деления на процедуры, подпрограммы и функции. В языке Си программа строится только из функций.

Функция — это независимая совокупность объявлений и операторов, обычно предназначенная для выполнения определенной задачи. Каждая функция должна иметь имя, которое используется для вызова функции. Имя одной из функций `main`, которая должна присутствовать в каждой Си-программе, зарезервировано. В программе могут содержаться и другие функции, причем функция `main` необязательно должна быть первой, хотя с нее всегда начинается выполнение Си-программы.

При вызове функции ей могут быть переданы данные посредством аргументов функции. Функция может возвращать значение. Это возвращаемое значение и есть основной результат выполнения функции, который при выполнении программы подставляется на место вызова функции, где бы этот вызов ни встретился в программе. Однако могут быть определены функции, которые не имеют никаких параметров и не возвращают никакого значения. Тем не менее действие таких функций может состоять в изменении внешних или статических переменных или выполнении каких-либо других действий, не связанных с данными.

С использованием функций в языке Си связаны три понятия: определение функции, объявление функции и вызов функции.

Определение функции задает имя функции, типы и число ее формальных параметров, объявления и операторы, которые определяют действие функции. Последовательность объявлений и операторов называется телом функции. В определении функции также может быть задан тип значения, возвращаемого функцией (тип возврата), и класс памяти.

Пример определения функции, проверяющей, является ли заданный символ русской буквой:

```
int ryc (unsigned char c)
{ if (c >= 'A' && c <= 'ë')
    return 1;
  else
    return 0;
}
```


Имя функции — рус. Функция имеет один параметр — исследуемый символ "с", который имеет тип `unsigned char`. Функция возвращает целое значение: 1, если символ является русской буквой, и 0, если нет.

В языке Си нет требования, чтобы определение функции обязательно предшествовало вызову функции. Определения используемых функций могут следовать за определением функции `main` или могут находиться вообще в другом файле.

Однако для того чтобы компилятор мог выполнить проверку соответствия типов передаваемых аргументов типам формальных параметров в определении функции, до вызова функции нужно поместить объявление (прототип) функции.

Прототип функции имеет такой же формат, что и определение функции, с той лишь разницей, что не имеет тела функции и что заголовок функции заканчивается точкой с запятой.

Прототип функции задает имя функции, типы и число формальных параметров, тип значения, возвращаемого функцией, и класс памяти. Формальные параметры могут иметь имена, но эти имена компилятору не нужны.

Прототип функции, определяющей, является ли символ русской буквой, может иметь следующий вид:

```
int рус (unsigned char );
```

В этом прототипе объявлено, что функция `рус` возвращает целое значение (`int`) и имеет один параметр типа `unsigned char`.

Компилятор использует прототип функции для сравнения типов фактических аргументов в вызове функции с типами формальных параметров.

Пример функции `main`, проверяющей, являются ли вводимые с консоли символы русскими буквами, с помощью функции `рус`:

```
#include <stdio.h>
unsigned char слово[20];
main()
{
    int рус(unsigned char );
    unsigned char ch;
    int i=0;
    while ((ch = getchar()) != (unsigned char) EOF)
    {
        if (рус(ch))
            слово[i++] = ch;
        else
            printf("\nНе буква %x\n", ch);
    }
    printf("%s\n%d\n", слово, i);
}
```


Прототипы библиотечных функций таких, как функции `getchar` и `printf` в данном примере, находятся в файлах включения, поставляемых в составе системы программирования и включаемых в программу с помощью директив препроцессора `#include`.

Если объявление функции не задано, то по умолчанию строится прототип на основании информации из первой ссылки на функцию, будь то вызов функции или определение. Однако такой прототип может неадекватно представлять последующий вызов или определение функции.

Рекомендуется всегда задавать прототип функции. Задание прототипов позволяет компилятору либо выдавать диагностические сообщения, либо корректным образом регулировать несоответствие аргументов, которое в противном случае не может быть установлено до выполнения программы.

Такая форма заголовка функции в определении или объявлении функции, когда объявление типов параметров находится в круглых скобках вслед за именем функции, соответствует проекту стандарта АНИС языка Си. В большинстве публикаций используется объявление параметров в так называемом "старом стиле", когда типы параметров объявляются перед телом функции, как в примере:

```
печать_таблицы (табл, счетчик)
  int *табл; /* Указатель на первый элемент массива */
  int счетчик; /* Число элементов */
{
  ...
}
```

Старый стиль удобен, когда для пояснения назначения параметров используются комментарии. Аналог прототипа функции, представленный в старом стиле, будем называть "синтаксической моделью". Такая синтаксическая модель не может быть включена в программу в качестве прототипа функции, но мы будем ею пользоваться при описании функции для удобства пояснения.

5.2. ОПРЕДЕЛЕНИЕ ФУНКЦИИ

Определение функции имеет следующий формат:

```
[ спецификатор-класса-памяти ] [ спецификатор-типа ]
  описатель ( [ список-формальных-параметров ] )
  тело-функции
```

Определение функции задает имя, формальные параметры и тело функции. Оно может также определять тип возвращаемого значения и класс памяти функции.

Класс памяти

Необязательный *спецификатор-класса-памяти* задает класс памяти функции, который может быть или `static`, или `extern`. Если он не задан, то по умолчанию считается `extern`. Подробная информация о классах памяти содержится в главе 6.

Тип возврата и имя функции

Необязательный *спецификатор-типа* и обязательный *описатель* задают тип возвращаемого функцией значения и имя функции.

Спецификатор типа может задавать любой основной тип, тип структуры или смеси. Если спецификатор типа не задан, то предполагается, что функция возвращает значение типа `int`.

Описатель функции может быть задан со звездочкой, что означает, что функция возвращает указатель. Функции не могут возвращать массив или функцию, но могут возвращать указатели на любой тип, включая массивы и функции.

Тип возвращаемого значения, задаваемый в определении функции, должен соответствовать типу в объявлениях этой функции, сделанных где-либо в программе.

Функция возвращает значение, если ее выполнение заканчивается выполнением оператора `return` в теле функции следующего формата:

`return` выражение;

Выражение вычисляется, преобразуется, если необходимо, к типу возвращаемого значения в определении (объявлении) функции, и управление возвращается в точку вызова.

Если оператор `return` не задан или не содержит выражения, функция не возвращает никакого значения. В этом случае спецификатор типа возвращаемого значения должен быть задан ключевым словом `void`, означающим отсутствие возвращаемого значения. Если функция объявлена со спецификатором типа возвращаемого значения, а фактически значение не возвращается, поведение вызывающей функции после возврата управления из такой функции может быть непредсказуемым.

Список формальных параметров

Список-формальных-параметров — это последовательность объявлений формальных параметров, разделенных запятыми.

Формальные параметры — это переменные, которые принимают значения, передаваемые функции во время вызова. Полное объявление формальных параметров функции задается в круглых скобках,

следующих за именем функции в определении функции. Элемент списка формальных параметров имеет два формата:

[register] спецификатор-типа [описатель]
[,...]

Список формальных параметров может заканчиваться запятой и многоточием (,...). Это означает, что число аргументов функции переменное. Однако предполагается, что функция имеет, по крайней мере, столько аргументов, сколько формальных параметров задано перед последней запятой. Функции может быть передано больше аргументов, чем задано до запятой и многоточия. Над такими аргументами не производится контроль типов.

Примечание. В качестве расширения проекта стандарта АНИС языка Си реализация допускает использование запятой (,) вместо запятой и многоточия (,...) в конце списка формальных параметров для задания переменного числа аргументов.

Если функции не передаются аргументы, то вместо списка формальных параметров необходимо задавать ключевое слово `void`.

Порядок и типы формальных параметров должны быть одни и те же в определении функции и во всех объявлениях (если они заданы). Типы фактических аргументов в вызовах функции должны быть совместимы с типами соответствующих формальных параметров. Формальный параметр может быть любого основного типа, структурой, смесью, перечислением, указателем или массивом.

Только класс памяти `register` можно задать для формального параметра. Если формальный параметр представлен в списке, но не объявлен, то предполагается, что параметр имеет тип `int`. Спецификатор типа может быть опущен, если задан класс памяти `register` для величины типа `int`.

Идентификаторы формальных параметров используются в теле функции в качестве ссылок на передаваемые величины. Эти идентификаторы не могут быть переопределены в блоке, объемлющем тело функции, но могут быть переопределены во внутренних блоках.

Если необходимо, компилятор выполняет обычные арифметические преобразования для каждого формального параметра и каждого фактического аргумента независимо. После преобразования нет формального параметра, короче чем `int`, и нет формального параметра с типом `float` (они преобразуются к типу `double`). Это значит, что объявить формальный параметр как `char` все равно, что объявить его как `int`.

Если используются ключевые слова `near`, `far`, `huge`, то компилятор выполняет преобразования аргументов-указателей функции, которые зависят от размера указателей и наличия или отсутствия списка типов аргументов.

Преобразованный тип каждого формального параметра определяет, как интерпретируются аргументы, помещаемые при вызове функции в стек. Несоответствие типов фактических аргументов и формальных параметров может быть причиной неверной интерпретации. Например, если 16-битовый указатель передается как фактический аргумент, а объявлен параметр типа `long`, то первые 32 бита стека проинтерпретируются как формальный параметр. Эта ошибка создает проблемы не только с формальным параметром типа `long`, но и с любыми формальными параметрами, которые следуют за ним. Ошибки такого рода могут быть обнаружены путем задания прототипов для всех функций.

Тело функции

Тело-функции — это составной оператор, содержащий операторы, определяющие действия функции. Он также может содержать объявления переменных, используемых в этих операторах.

Все переменные, объявленные в теле функции, имеют класс памяти `auto`, если они не объявлены иначе, и являются локальными. Когда вызывается функция, то для локальных переменных отводится память в стеке и производится их инициализация (если она задана). Управление передается первому оператору составного оператора, и начинается процесс выполнения, который продолжается до тех пор, пока не встретится оператор `return` или конец тела функции. Управление при этом возвращается в точку вызова.

Если функция возвращает значение, то должен быть выполнен оператор `return`, содержащий выражение. Значение возврата не определено, если оператор `return` не выполнен или если в операторе `return` не было указано выражение.

Как уже было сказано, параметры функции передаются по значению и могут рассматриваться как локальные переменные, место для которых распределяется при вызове функции, они инициализируются значениями переданных аргументов и теряются при выходе из функции. Поэтому в теле функции нельзя изменить значение параметра, так как функция работает с копиями аргументов. Однако, если в качестве параметра передать указатель переменной, можно использовать операцию разадресации этой переменной для осуществления доступа и изменения ее значения.

Рассмотрим примеры неправильного и правильного определения функции, обменивающей значение двух переменных.

```
/* Неправильная функция обмена */
void обмен (int x, int y)
{
    int врем = x;
    x=y;
    y=врем;
}
```


Так как параметры *x* и *y*, принимаемые функцией, являются копиями передаваемых параметров, она обменивает значение этих параметров, не оказывая влияние на значение аргументов, и эта функция не выполняет своей задачи. Правильная работа функции обеспечивается при таком ее определении:

```
/* Правильная функция обмена */  
void обмен (int *ук_x, int *ук_y)  
{  
    int врем=*ук_x;  
    *ук_x=*ук_y;  
    *ук_y=врем;  
}
```

При вызове функции ей должны передаваться в качестве аргументов адреса переменных *x* и *y*, значения которых нужно обменять.

5.3. ОБЪЯВЛЕНИЕ ФУНКЦИИ (ПРОТОТИП)

Если нужно вызвать функцию до ее определения в этом файле или определение функции находится в другом исходном файле, вызов функции следует предварить объявлением функции, т.е. задать прототип функции, который позволит компилятору выполнить проверку аргументов и возвращаемого значения. Прототип функции имеет следующий формат:

```
[ спецификатор-класса-памяти] [спецификатор-типа]  
    описатель([список-формальных-параметров])  
    [,список-описателей];
```

В отличие от формата определения функции в формате прототипа за заголовком функции следует точка с запятой (т.е. прототип функции не имеет тела), и заголовок функции может заканчиваться *списком-описателей*. Список описателей представляет имена других функций, возвращающих значение того же типа, который задан спецификатором типа для первого описателя функции, или других переменных такого же типа.

Правила использования остальных элементов формата прототипа те же, что и в определении функции. Имена параметров в списке формальных параметров указывать нет необходимости. Такие имена имеют область действия только до конца объявления, и никакой смысловой нагрузки не несут.

Прототип — это явное объявление функции, которое предшествует определению функции. Тип возврата, заданный в прототипе, должен согласовываться с типом возврата, заданным в определении.

Если прототип функции не задан, а встретился вызов функции, компилятор создает неявный прототип, исходя из информации, доступной из вызова функции. Тип возврата создаваемого прототипа функции `int`. На основании типов и числа фактических аргументов создается список объявлений формальных параметров.

Таким образом, прототипы функций нужно задавать в следующих случаях:

1) функция возвращает значение типа, отличного от `int` (хотя хороший стиль программирования требует задания прототипа и для случая `int`);

2) указатель на функцию должен быть проинициализирован до того, как функция будет определена. Это обеспечивается заданием прототипа функции.

Наличие в прототипе полного списка типов параметров позволяет компилятору выполнить проверку соответствия типов аргументов в вызове функции или ее определении типам, указанным в прототипе, или выполнить соответствующее преобразование. В прототипе может быть отражен и тот факт, что число аргументов переменное или что аргументы вообще не передаются.

Если прототип задан с классом памяти `static`, в определении функции также должен быть задан спецификатор класса памяти `static`. Если задан спецификатор класса памяти `extern` или спецификатор опущен, то функция имеет класс памяти `extern`.

5.4. ВЫЗОВЫ ФУНКЦИЙ

Вызов функции имеет следующий формат:

выражение ([список-выражений])

Выражение вычисляется как адрес функции. *Список-выражений* представляет собой список фактических аргументов, передаваемых в функцию. Он может быть пустым.

Фактический аргумент может быть любой величиной основного типа, структурой, перечислением, смесью или указателем. Хотя массивы и функции не могут быть переданы как параметры, но указатели на эти объекты могут быть переданы.

Выполнение вызова функции происходит следующим образом.

1. Вычисляются выражения в списке выражений и производятся обычные арифметические преобразования. Затем, если известен прототип функции, тип результирующего аргумента сравнивается с типом соответствующего формального параметра. Если они не совпадают, то либо производится преобразование типов, либо выдается диагностическое сообщение.

Число выражений в списке выражений должно совпадать с числом формальных параметров, если только прототип функции или определение не определяет явно переменное число аргументов. В этом случае компилятор проверяет столько аргументов, сколько имен типов имеется в списке формальных параметров и преобразует их, если необходимо, по описанным правилам.

Если в прототипе функции вместо списка формальных параметров задано ключевое слово `void`, это значит, что в функцию не передается никаких аргументов и в определении функции не должно быть формальных параметров. Если это не так, то выдается диагностическое сообщение.

2. Происходит замена формальных параметров на фактические. Первое выражение в списке всегда соответствует первому формальному параметру, второе — второму и т.д.

3. Управление передается на первый оператор функции.

4. Выполнение оператора `return` в теле функции возвращает управление и, возможно, значение в вызывающую функцию. Если оператор `return` не задан, то управление возвращается после выполнения последнего оператора тела функции. При этом возвращаемое значение не определено.

Примечание. Выражения в списке аргументов могут вычисляться в любом порядке, поэтому аргументы, зависящие от побочных эффектов вычисления других аргументов, могут иметь в разных реализациях разные значения. Точка последовательности, устанавливаемая операцией вызова функции, гарантирует только, что все побочные эффекты будут вычислены перед передачей управления в вызванную функцию.

Как сказано выше, выражение перед скобками вычисляется как адрес функции. Это означает, что функцию можно вызвать через указатель на функцию.

Например, в объявлении

```
int (* func_ptr) (int, int );
```

идентификатор `func_ptr` объявлен как указатель на функцию с двумя аргументами типа `int` и возвращающую значение типа `int`. Вызов функции в этом случае может выглядеть так:

```
(*func_ptr) (i,j);
```

Здесь используется операция разадресации (*), чтобы получить адрес функции, на которую ссылается указатель `func_ptr`. Адрес функции используется для ее вызова.

Круглые скобки, заключающие `*func_ptr`, обязательны, так как в противном случае будет иметь место объявление функции `func_ptr`, возвращающей указатель на целое.

Указатель на функцию может быть передан в качестве параметра функции. Указатель на функцию разадресуется в вызов функции, на

которую он указывает. Чтобы присвоить значение указателю на функцию в выражении присваивания, нужно употребить имя функции без списка параметров. Пусть, например, имеется объявление:

```
double (*func_ptr) (int,int), power(int,int);
```

После присваивания

```
func_ptr=power;
```

указатель `func_ptr` будет указывать на функцию `power` и вызов

```
(*func_ptr) (2,5);
```

будет эквивалентен вызову

```
power(2,5);
```

Рассмотрим использование указателя на функцию в качестве параметра библиотечной функции линейного поиска `lfind`, поставленной в составе библиотеки Си. Синтаксическая модель этой функции:

```
char *lfind (key,base,num,width,compare);
char *key; /* Указатель ключа поиска */
char *base; /* Указатель базы поиска */
unsigned *num, width; /*Количество и длина элементов*/
int (*compare) (elem1,elem2); /* Указатель на пользователь- */
/* скую функцию сравнения */
const void *elem1, *elem2; /* Элементы для сравнения */
```

Пример.

```
/* Поиск элементов в массиве */
```

```
#include <search.h>
```

```
#include <string.h>
```

```
#include <stdio.h>
```

```
int compare(char *, char *);
```

```
struct сведения
```

```
{
```

```
    char фιο[20];
```

```
    long телефон;
```

```
} справочник[] = {
```

```
    {"Иванов И.И. ", 152374 },
```

```
    {"Петров П.П. ", 231415 },
```

```
    {"Сидоров С.С.", 441277 }
```

```
};
```

```
char *key = "Петров П.П. ";
```

```
main()
```

```
{
```

```
    struct сведения *result;
```

```
    unsigned num, width;
```

```
    width = sizeof ( struct сведения);
```

```
    num = sizeof (справочник) / width ;
```

```
    result = (struct сведения *) lfind (key, (char *)справочник,
                                     &num, width, compare);
```

```
    if (result)
```

```
        printf("Номер телефона абонента %20s %ld\n",
               key, result->телефон);
```



```

else
    printf("Абонент %20s не найден\n", key);
}
int compare(char *arg1, char *arg2)
{
    return(strncmp(arg1, arg2, strlen(arg1)));
}

```

Эта программа отыскивает в массиве структур, содержащем фамилии и номера телефонов абонентов, абонента "Петров П.П." и печатает его номер телефона. Аргументом функции `lfind`, которая отыскивает фамилию "Петров П.П.", является указатель на функцию `compare`, которая сравнивает два элемента поиска. Прототип функции `compare` должен быть указан до использования идентификатора функции в качестве параметра, иначе идентификатор не будет распознан.

Рекурсивные вызовы

Любая функция в программе на языке Си может быть вызвана рекурсивно, т.е. она может вызвать саму себя. Компилятор допускает любое число рекурсивных вызовов. При каждом вызове для формальных параметров и переменных с классом `auto` и `register` выделяется новая память, так что их значения из предшествующих незавершенных вызовов не затираются. Параметры предшествующих незавершенных вызовов недоступны.

Классический пример рекурсии — это математическое определение факториала $n!$:

$$n! = \begin{cases} 1 & n=0 \\ n*(n-1)! & n>1 \end{cases}$$

Функция для вычисления факториала для $n \geq 0$:

```

long fact(int n)
{
    return (n<=1) ? 1 : n*fact(n-1);
}

```

Переменные, объявленные с классом памяти `static`, не требуют новой памяти при каждом рекурсивном вызове. Их память сохраняется на все время выполнения программы.

Хотя компилятор языка Си не ограничивает число рекурсивных вызовов функции, операционная среда может накладывать практические ограничения. Так как каждый рекурсивный вызов требует дополнительной стековой памяти, то слишком много рекурсивных вызовов могут переполнить стек.

Вызовы функций с переменным числом аргументов

Чтобы вызвать функцию с переменным числом аргументов, в вызове функции просто задается любое число аргументов. В объявлении (прототипе) и определении функции переменное число аргументов задается запятой с последующим многоточием (,...) в конце списка формальных параметров или списка типов аргументов.

Все аргументы, заданные в вызове функции, размещаются в стеке. Количество формальных параметров, объявленных для функции, определяет число аргументов, которые берутся из стека и присваиваются формальным параметрам. Программист отвечает за выбор дополнительных аргументов из стека и определение того, сколько аргументов находится в стеке.

Примерами функций с переменным числом параметров являются функции из библиотеки функций Си: `printf`, `fprintf`, `sprintf` для форматизованного вывода любого числа данных или функции `scanf`, `fscanf`, `sscanf` для форматизованного ввода любого числа данных. Прототипы функций форматизованного вывода, например, имеют такой вид:

```
int printf (const char * формат [,аргумент],...);
int fprintf (FILE *поток, const char *формат [,аргумент],...);
```

Для этих и других функций библиотеки Си с переменным числом параметров выполняется только ограниченная проверка типов аргументов. Так, для функций `printf` и `scanf` проверка типа выполняется только для первого аргумента (строки формата), а для функций типа `fprintf`, `fscanf` проверяются типы только двух первых аргументов (файла и строки формата).

Мобильный способ обработки вызовов с переменным числом параметров

Пользователь может разработать свои собственные функции с переменным числом параметров. Для обеспечения мобильного способа доступа к аргументам функции, когда функция имеет переменное число аргументов, в библиотеке функций Си имеются три функции `va_start`, `va_arg` и `va_end`, представляющие собой макроопределения и, следовательно, находящиеся в файле включения, а именно, в файле `stdarg.h`. Эти макроопределения предполагают, что функция имеет некоторое число обязательных аргументов, за которыми следует переменное число необязательных аргументов. Обязательные аргументы доступны через имена параметров как обычные параметры функции. Чтобы извлечь необязательный аргумент, нужно использовать макрокоманды `va_start`, `va_arg` и `va_end` в определенном порядке. Макро-

команда `va_start` предназначена для установки аргумента `arg_ptr` на начало списка необязательных аргументов и имеет вид:

```
void va_start(arg_ptr, prev_param);
```

Параметр `prev_param` представляет параметр, предшествующий первому необязательному параметру.

Макрокоманда `va_arg` обеспечивает доступ к текущему аргументу и имеет вид:

```
type va_arg (arg_ptr, type);
```

Параметр `type` представляет тип аргумента. Макрокоманда `va_end` устанавливает указатель `arg_ptr` списка необязательных параметров по окончании обработки всех необязательных аргументов на нуль (`NULL`).

Указатель `arg_ptr` должен быть объявлен с предопределенным в системе именем типа `va_list`:

```
va_list arg_ptr;
```

Макрокоманда `va_start` должна быть использована до первого использования макрокоманды `va_arg`. Макрокоманда `va_arg` извлекает значение типа `type` из ячейки памяти, заданной аргументом `arg_ptr`, и увеличивает указатель `arg_ptr`, чтобы он указывал на следующий аргумент в списке, при этом размер типа `type` используется для определения начала следующего аргумента. Макрокоманда `va_arg` используется столько раз, сколько необходимо для извлечения всех аргументов из списка.

Рассмотрим применение этих макрокоманд для обработки параметров функции "среднее_знач" с переменным числом аргументов. Признаком конца списка аргументов является значение "-1".

```
#include <stdio.h>
#include <stdarg.h>
main()
{
    int n, среднее_знач (int,...);
    n = среднее_знач(2,3,4,-1);          /* Вызов с 4 аргументами */
    printf ("Среднее значение: %d\n", n);
    n = среднее_знач (5,6,7,8,9,-1);    /* Вызов с 6 аргументами */
    printf ("Среднее значение: %d\n", n);
}
int среднее_знач (int первый ,...)
{
    int i=0, счет=0, сумма=0;
    va_list ук_arg;
    va_start (ук_arg, первый);
    if (первый != -1)
        сумма = первый;
    else
        return (0);
```



```

счет++;
for (; (i=va_arg(yк_apr, int)) >=0;)
{
    сумма += i;
    счет++;
}
va_end(yк_apr);
return (сумма/счет);
}

```

Передача параметров из операционной системы

Функция `main`, с которой начинается выполнение Си-программы, может быть определена с параметрами, которые передаются из внешнего окружения, например из командной строки.

Принято, что два первых параметра функции `main` называются `argc` и `argv`, хотя это не диктуется языком Си.

Параметр `argc` определяет общее число аргументов, передаваемых функции `main`, и объявляется как `int`. Параметр `argv` объявляется как массив указателей, каждый элемент которого ссылается на строковое представление аргумента, передаваемого функции `main`. Если функции `main` передается третий параметр, этот параметр принято называть `envp`. Параметр `envp` объявляется аналогично параметру `argv` как массив указателей на строковые величины, определяющие операционную среду, в которой выполняется программа.

Аргумент `argv[0]` всегда содержит имя командной строки, поэтому значение `argc` на единицу превышает количество передаваемых аргументов. Аргументы командной строки могут задавать, например, режимы работы, имена файлов и другие данные для программы и отделяются друг от друга пробельными символами. Если пробельный символ должен быть представлен в аргументе, аргумент заключается в кавычки.

Полный заголовок функции `main` имеет вид:

```

main (argc, argv, envp)
int argc;          /* Число аргументов, переданное функции main, плюс 1 */
char *argv[];      /* Массив указателей строковых величин, представляющих
                    аргументы, передаваемые функции main */
char *envp[];      /* Массив указателей строковых величин, определяющих
                    операционную среду */
{

```

Если, например, командная строка вызова Си-программы имеет вид:

```
A:\>cprog Working C program
```

то аргументы `argc`, `argv` и `envp` представляются в памяти, как показано на рис. 3.

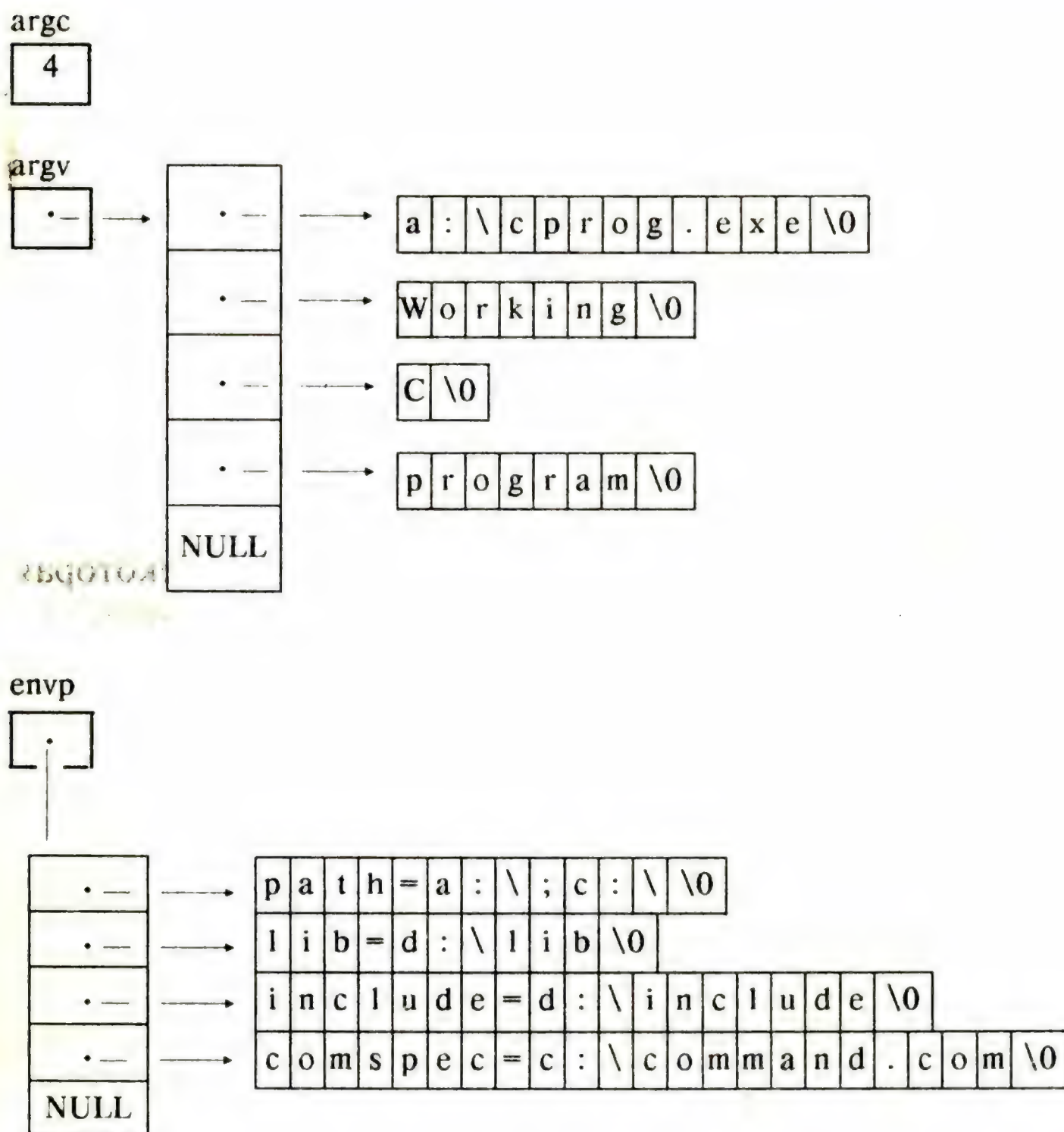


Рис. 3. Представление аргументов командной строки в памяти

Операционная система поддерживает передачу значений для параметров `argc`, `argv` и `envp`, а на пользователе лежит ответственность за передачу фактических аргументов функции `main`. Соглашение о передаче параметров в большей степени определяется операционной системой, чем самим языком Си. Формальные параметры должны быть объявлены в определении функции.

Следующий пример представляет программу печати значения фактических аргументов, передаваемых функции `main`, и переменных операционной среды:

```
#include <stdio.h>
main (int argc, char *argv[], char *envp[])
{
    printf("Имя программы и передаваемые аргументы:\n");
    while(*argv)
```



```

    printf("%s\n",*argv++);
    printf("\nПеременные среды:\n");
    while(*envp)
        printf("%s\n",*envp++);
}

```

Для получения информации о переменных среды в библиотеке функций предоставляется функция `getenv`, имеющая прототип:

```
char *getenv (const char *varname);
```

где `varname` задает имя переменной среды, указатель на значение которой должна выдать функция `getenv`. Если указанная переменная не определена в среде в данный момент, то возвращается значение `NULL`.

Для изменения значения переменной среды может использоваться функция `putenv`.

Перед выполнением любой Си-программы выполняется некоторая инициализация, которая включает, например, и такие действия, как обработка аргументов, передаваемых программе, и переменных среды. Эти действия выполняются библиотечными функциями `_setargv` и `_setenvp`, которые всегда включаются компилятором в Си-программу перед функцией `main`. Если же в программе параметры `argv` и `envp` никогда не используются (нет передаваемых аргументов и переменные операционной среды не используются), то можно запретить использование библиотечных функций `_setargv` и `_setenvp`. Для этого надо в файле, содержащем функцию `main`, определить функцию `_setargv` и `_setenvp`, которые ничего не делают. Например, так:

```

_setargv ()
{
}
_setenvp ()
{
}
main ()
{
...

```

Тогда при вызове библиотечных функций `_setargv` и `_setenvp` будут использоваться функции, определенные пользователем. Это несколько улучшит эффективность программы.

5.5. СТАНДАРТНЫЕ ФУНКЦИИ БИБЛИОТЕКИ СИ

Система программирования Си предоставляет программисту ряд предварительно определенных функций и макроопределений, предназначенных для использования в программах на языке Си. Большинство функций доступны программисту в откомпилированном виде и подключаются к Си-программе во время редактирования связей. Часть функций представляют собой макроопределения и поэтому дол-

жны быть доступны во время компиляции. Это обеспечивается включением в исходный файл соответствующего файла. Некоторые функции реализованы и как макроопределения, и как функции. Используемый вариант определяется в зависимости от того, включен ли в программу файл с макроопределением или нет. Основным преимуществом использования макрокоманд является более быстрое время выполнения. Макрокоманды на этапе препроцессирования заменяются макрорасширениями и код, требуемый для вызова функции, не генерируется. Однако макрокоманды увеличивают размер программы, особенно если макрокоманда встречается в программе несколько раз. Каждое появление макрокоманды заменяется макрорасширением, в отличие от функции, которая определяется только один раз, независимо от числа вызовов.

Другие отличия макроопределений от функций, которые следует учитывать программисту, состоят в следующем:

- имя макроопределения в отличие от имени функции не вычисляется как адрес и поэтому не может использоваться в контексте, требующем указатель;

- макроопределение нельзя объявить и поэтому нельзя проверить типы аргументов.

Для всех функций библиотеки Си, не являющихся макроопределениями, предоставляются файлы включения, содержащие объявления этих функций (прототипы). Эти файлы должны включаться в программу, чтобы обеспечить возможность проверки типов аргументов. Несоответствие типов аргументов функции формальным параметрам может привести к серьезным ошибкам, которые трудно обнаружить.

Предоставляемые программисту функции библиотеки можно разделить на несколько категорий в соответствии с их назначением.

Проверка и преобразование символов. Функции этой группы выполняют проверку символов на принадлежность соответствующему классу (алфавитно-цифровых символов, букв, управляющих символов, десятичных цифр, шестнадцатеричных цифр, печатных символов, строчных букв, прописных букв, пробельных символов, знаков пунктуации), а также преобразование букв из строчных в прописные и наоборот.

Преобразование данных. Функции этой группы выполняют преобразование числовых данных из строкового представления в число типа `int`, `long`, `double`, `float` и наоборот.

Операции над строками. Эти функции выполняют сравнение строк, поиск отдельных символов или символов из заданного набора в строке, копирование строк, преобразование символов в прописные или строчные, заполнение строки заданным символом, реверсирование строки, разбиение строки на лексемы, соединение строк, нахождение

длины строки. Во всех этих операциях предполагается, что строка заканчивается нулевым символом.

Операции над буферами. В отличие от операций над строками операции над буферами не требуют наличия нулевого символа в конце операнда. Эти функции всегда получают длину операнда в качестве аргумента. Операции над буферами аналогичны основным операциям над строками.

Математические функции. Это тригонометрические и гиперболические функции, функции Бесселя, функции преобразования в представление IEEE чисел с плавающей точкой, вычисление логарифма, получение слова состояния арифметического сопроцессора.

Поиск и сортировка. Эти функции обеспечивают бинарный и линейный поиск и быструю сортировку.

Распределение памяти. Эти функции позволяют получить, освободить и перераспределить блок памяти из ресурсов операционной системы или из области динамически распределяемой памяти программы ("кучи"). Для различных моделей памяти программы имеются соответствующие функции распределения памяти.

Управление процессами. Процесс — это программа, которая запускается операционной системой. Используя функции управления процессами, можно запустить процесс и из программы. Функции управления процессами позволяют идентифицировать процесс уникальным номером, начать новый процесс, завершить процесс, обработать сигнал прерывания, послать сигнал процессу, выполнить функции при завершении процесса, выполнить команду Альфа-ДОС. Имеется несколько разновидностей функций выполнения порожденного процесса, различающихся способом передачи аргументов порожденному процессу, использованием установки среды родительского процесса или передачей порожденному процессу таблицы установки переменных среды, использованием текущей установки переменной MARSH (PATH) для нахождения файла, который будет выполняться как процесс.

Управление каталогом. Эти функции позволяют обеспечить доступ, модификацию и получение информации о каталогах системы.

Управление файлами. Эти функции работают либо с именем файла, либо с дескриптором файла, представляющим структуру управления файлом, возвращаемую операционной системой при открытии файла. Функции управления файлами позволяют получить информацию о файле, проверить и изменить режим доступа и размер файла, удалить и переименовать файл.

Ввод-вывод. Функции этой категории обеспечивают ввод-вывод потоком, ввод-вывод нижнего уровня, ввод-вывод для консоли и портов и подробнее рассматриваются в главе 11.

Вызовы операционной системы. Эти функции обеспечивают интерфейс с BIOS посредством выдачи соответствующих прерываний, а также интерфейс с операционной системой Альфа-ДОС посредством системных вызовов.

Графические функции. Графические функции делятся на три подкатегории: графические примитивы, функции работы со шрифтами и функции вывода диаграмм. Графические примитивы позволяют построить графические изображения на экране дисплея, обеспечивающего графические возможности, с использованием различных палитр и перемещать их (подробнее в главе 13). Функции работы со шрифтами поддерживают следующий набор шрифтов для вывода текста: Курьер, Гельветика, Таймс, Модерн, Скрипт, Романский, а функции вывода диаграмм дают возможность строить диаграммы нескольких типов: в виде одно- или многорядной полосы или столбца, линейной, точечной или круговой диаграммы.

Функции работы с окнами. Эти функции позволяют создавать простые или стековые окна на экране дисплея (причем окна могут накладываться друг на друга). Окна могут использоваться для просмотра текстовых файлов, ввода и вывода информации, организации работы с меню и выполнять другие функции, специфичные для окон (подробнее в главе 12).

Перечень основных функций библиотеки с указанием их прототипов приведен в приложении 2.

Глава 6

СТРУКТУРА ПРОГРАММЫ И КЛАССЫ ПАМЯТИ

6.1. ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ

Обычная Си-программа представляет собой определение функции `main`, которая для выполнения необходимых действий вызывает другие функции. Примеры программ, которые рассматривались до сих пор, представляли один исходный файл, содержащий определение функции `main` и функций, вызываемых из `main` (исключая функции библиотеки Си, которые в большинстве своем доступны программисту в откомпилированном, объектном виде). Эти функции осуществляют связь по данным посредством передачи параметров и возврата значений.

Однако реальный программный проект разрабатывается как множество исходных файлов, которые компилируются отдельно, а затем объединяются в исполняемую программу редактором связей. При та-

кой структуре исходной программы функции из разных файлов могут использовать глобально доступные внешние переменные. Сами функции по определению всегда внешние и доступны из любых файлов.

Пример скелета программы, состоящей из двух файлов file1.C и file2.C, представлен на рис. 4.

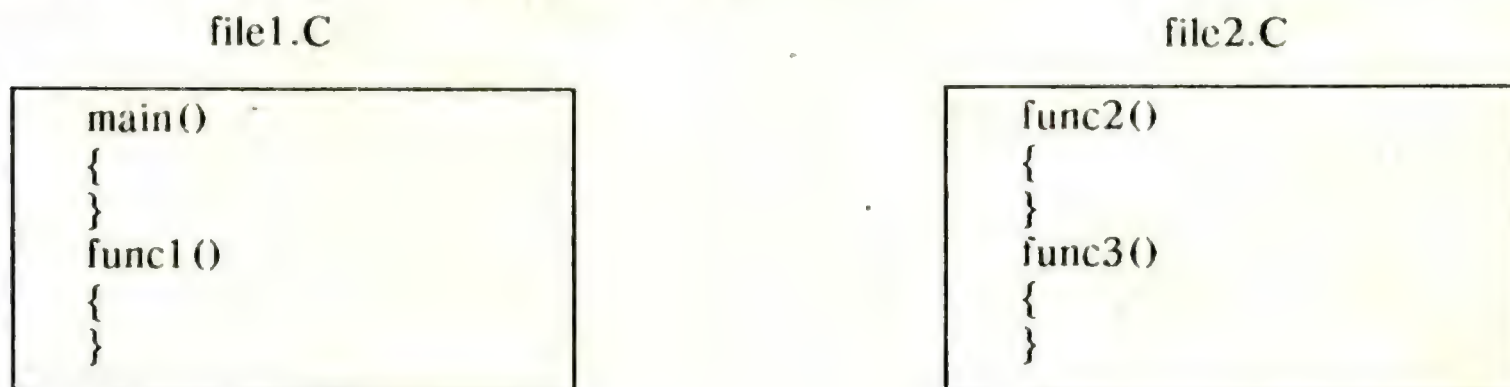


Рис. 4. Пример скелета программы, состоящей из двух исходных файлов

В этом примере функция main может выполнить любую из трех функций func1, func2, func3. Каждая из этих функций может вызывать любую из двух оставшихся.

Для того чтобы определяемая функция могла выполнить какую-нибудь работу, она должна использовать переменные. В языке Си все переменные должны быть объявлены до их использования. Объявления устанавливают соответствие между именем и атрибутами переменной, функции либо типа. Определение переменной вызывает выделение памяти для переменной.

Класс памяти, выделяемой для переменной, определяется спецификатором класса памяти в определении переменной.

Класс памяти переменной определяет время жизни и область видимости переменной, связанные с понятием блока программы.

Блоком в программе на языке Си является последовательность объявлений, определений и операторов, заключенная в фигурные скобки. Существуют два типа блоков:

- 1) составной оператор;
- 2) определение функции, состоящее из составного оператора, включающего в себя тело функции, плюс заголовок функции (т.е. имя функции, тип возвращаемого значения и формальные параметры).

Блоки могут включать в себя составные операторы, но не могут содержать определение функции. Внутренний блок называется вложенным, а внешний блок — объемлющим.

Примечание. Все составные операторы заключаются в фигурные скобки, но не всякий объект, заключенный в фигурные скобки, является составным оператором. Например, инициаторы определения смеси, структуры и перечисления включают фигурные скобки, но они не считаются составными операторами.

Время жизни — это интервал выполнения программы, в течение которого программный объект (переменная или функция) существует.

Время жизни переменной может быть глобальным или локальным. Объект с глобальным временем жизни имеет распределенную для него память и определенное значение на протяжении всего выполнения программы. Объект с локальным временем жизни имеет распределенную для него память и определенное значение только внутри блока, где объект определен или объявлен. Для локального объекта распределяется новая область памяти при каждом входе в блок и освобождается (а, следовательно, значение объекта теряется) при выходе из блока.

Функции в программе имеют глобальное время жизни, т.е. существуют на протяжении всего выполнения программы.

Область видимости — часть текста программы, в которой может быть использован данный объект. Объект считается видимым в блоке или исходном файле, если известны тип и имя объекта в блоке или исходном файле. Объект может быть видимым в пределах блока, одного исходного файла или всех исходных файлов, образующих программу (т.е. глобально видим), в зависимости от того, на каком уровне объект объявлен: внутреннем (внутри блока) или внешнем (вне всех блоков программы). Если объект объявлен внутри блока, он видим только в этом блоке и блоках, вложенных в него. Если объект объявлен на внешнем уровне, он видим от точки объявления до конца данного файла.

Объект может быть сделан глобально видимым с помощью соответствующих объявлений во всех исходных файлах, образующих программу.

Спецификатор класса памяти в объявлении переменной может быть `auto`, `register`, `extern` и `static`. Если класс памяти не указан, он определяется по умолчанию из контекста объявления.

Объекты классов `auto` и `register` имеют локальное время жизни. Спецификаторы `static` и `extern` определяют объекты с глобальным временем жизни. Точный смысл каждого спецификатора класса памяти зависит от того, находится ли объявление на внешнем или внутреннем уровне, и от того, является ли объект функцией или переменной.

6.2. ОБЪЯВЛЕНИЕ ПЕРЕМЕННЫХ НА ВНУТРЕННЕМ УРОВНЕ

На внутреннем уровне при объявлении переменной может быть использован любой из четырех спецификаторов класса памяти. Если при объявлении переменной на внутреннем уровне спецификатор класса памяти опущен, то подразумевается класс памяти `auto`.

Переменная с классом памяти `auto` имеет локальное время жизни. Она видима только в том блоке, в котором объявлена. Память для

переменной класса `auto` выделяется при входе в блок и освобождается при выходе из блока. При повторном входе в блок этой переменной может быть распределено другое место в памяти.

Переменная с классом памяти `auto` автоматически не инициализируется. Она должна быть проинициализирована явно при объявлении или путем присваивания ей начального значения операторами языка Си. Значение неинициализированной переменной с классом памяти `auto` считается неопределенным.

Спецификатор класса памяти `register` предписывает компилятору распределить память под переменную в регистре, если это возможно. Использование регистровой памяти обычно приводит к более быстрому времени доступа и к меньшему размеру результирующего кода. Переменные, объявленные с классом памяти `register`, имеют ту же область видимости, что и переменные `auto`. Число регистров, которое может быть использовано под память переменных, зависит от машины. Когда компилятор встречает спецификатор класса памяти `register` в объявлении, а свободного регистра не имеется, то для переменной распределяется память класса `auto`. Компилятор назначает переменным регистровую память в том порядке, в котором появляются объявления в исходном файле. Регистровая память, если она имеется, может быть назначена только для типов `int` и указателей, которые имеют такой же размер, что и `int`.

Переменные, объявленные на внутреннем уровне со спецификатором класса памяти `static`, обеспечивают возможность сохранить значение локальной переменной при выходе из блока и использовать его при последующем входе в блок.

Переменная, объявленная на внутреннем уровне со спецификатором класса памяти `static`, имеет глобальное время жизни и видимость только внутри блока, в котором она объявлена. В отличие от переменных `auto`, память для которых распределяется в стеке, для переменных, объявленных как `static`, память отводится в сегменте данных, благодаря чему они сохраняют свое значение при выходе из блока.

Пример объявления переменной "счетчик" на внутреннем уровне с классом памяти `static` приведен на рис. 5.

file1.C

```
main()
{
}
func1()
{
    static int счетчик=0;
}
```

file2.C

```
func2()
{
    static int счетчик=0;
}
func3()
{
    static int счетчик=0;
}
```

Рис. 5. Объявление статической переменной на внутреннем уровне

В примере на рис. 5 определены три статистических переменных с одинаковым именем "счетчик", каждая из которых имеет глобальное время жизни, но видима только в той функции, в которой она определена. Переменная "счетчик" может использоваться, например, для подсчета числа обращений к функции.

Переменные класса памяти `static` могут быть инициализированы константным выражением. Если явной инициализации нет, то переменная класса памяти `static` устанавливается в 0. Инициализация выполняется один раз и не повторяется при новом входе в блок. Переменные класса памяти `static` могут быть инициализированы адресом любого внешнего объекта, но только не адресом объекта с классом памяти `auto`, так как адрес объекта с классом памяти `auto` не является постоянным во время выполнения программы. Он изменяется при каждом входе в блок.

Переменная, объявленная со спецификатором класса памяти `extern`, является ссылкой на переменную с тем же самым именем, определенную на внешнем уровне в любом исходном файле программы. Цель внутреннего объявления `extern` состоит в том, чтобы сделать определение переменной внешнего уровня видимым внутри блока. Не будучи объявленной на внешнем уровне, переменная, объявленная как `extern`, видима только в блоке, в котором она объявлена.

Пример объявления переменной "счетчик", являющейся именем внешнего массива целых чисел типа `long`, на внутреннем уровне представлен на рис. 6.

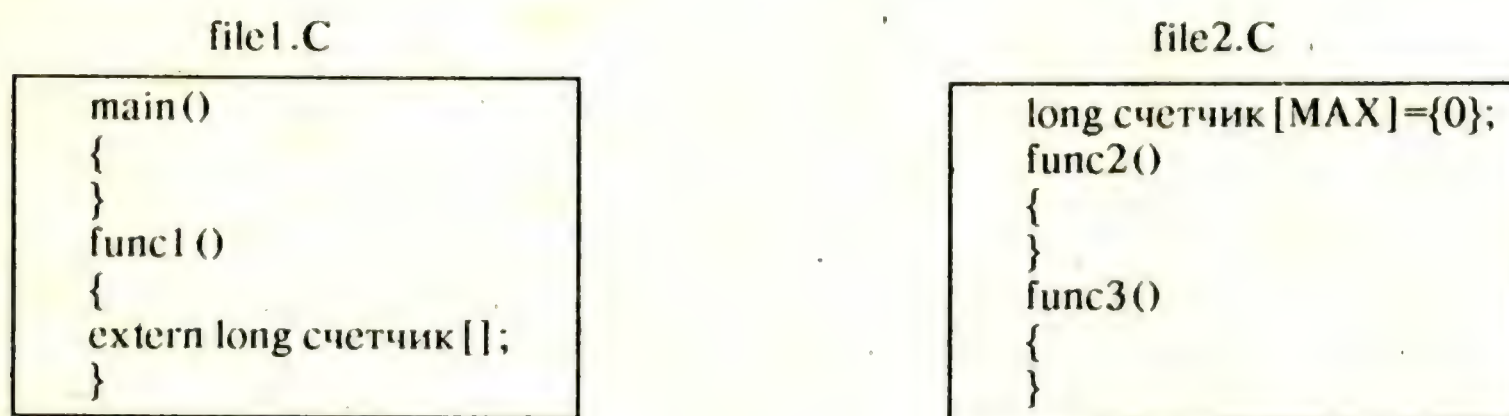


Рис. 6. Объявление внешней переменной на внутреннем уровне

Объявление переменной "счетчик[]" как `extern` в примере на рис. 6 делает ее доступной в функции `func1`. Определение этой переменной находится в файле `file2.C` на внешнем уровне и должно быть только одно, в то время как объявлений как `extern` может быть несколько (например, и в функции `main`). Даже если определение внешней переменной находится в этом же файле, но позже, чем возникает необходимость в ее использовании, объявление `extern` должно быть помещено в этот файл до использования переменной (рис. 7).

Объявление переменной со спецификатором `extern` определяет тип переменной и информирует компилятор о том, что память для этой переменной распределяется где-то в другом месте.

file1.C

```
main( )
{
}
func1( )
{
extern long счетчик [ ];
}
static long счетчик [MAX]={0};
```

Рис. 7. Объявление внешней переменной, определенной в этом же файле

6.3. ОБЪЯВЛЕНИЕ ПЕРЕМЕННЫХ НА ВНЕШНЕМ УРОВНЕ

При объявлении переменных на внешнем уровне могут использоваться только спецификаторы класса памяти `static` и `extern` или переменная объявляется без спецификатора класса памяти. На внешнем уровне не допускаются спецификаторы класса памяти `auto` и `register`.

Объявления переменных на внешнем уровне — это или определения переменных, или ссылки на определения, сделанные в другом месте. Объявление внешней переменной, которое инициализирует эту переменную (явно или неявно), является определением этой переменной. Определение на внешнем уровне может задаваться в следующих различных формах:

1) переменная на внешнем уровне может быть определена путем ее объявления со спецификатором класса памяти `static`. Такая переменная может быть явно инициализирована константным выражением. Если инициализация отсутствует, то переменная автоматически инициализируется нулем по умолчанию. Таким образом, объявления:

```
static int i=0;
```

и

```
static int i;
```

рассматриваются как определения;

2) переменная определяется, когда она явно инициализируется на внешнем уровне. Например,

```
int i=3;
```

является определением переменной "i".

Так как переменная определяется на внешнем уровне, то она види-

ма в пределах остатка исходного файла, в котором она определена. Переменная невидима выше своего определения ни в том же самом исходном файле, ни в других исходных файлах программы, если не объявлена ссылка, которая делает ее видимой.

Так, в примере на рис. 6 определение переменной "счетчик [MAX]" в файле file2.C, расположенное в начале файла, обеспечивает видимость этой переменной в функциях func2 и func3 без дополнительных объявлений ее как extern, в то время как определение этой переменной в файле file1.C на рис. 7 не обеспечивает ее видимость в функции main.

Переменная может быть определена на внешнем уровне внутри исходного файла только один раз. Если задается спецификатор класса памяти static, то в других исходных файлах могут быть определены переменные с тем же именем. Так как каждое определение static видимо только в пределах своего исходного файла, то конфликта не возникает.

Спецификатор класса памяти extern используется для объявления ссылки на переменную, определенную где-то в другом месте. Такие объявления используются в случае, когда нужно сделать видимым объявление переменных в других исходных файлах или выше места, где переменная определена в том же самом исходном файле. Так как ссылка на переменную объявлена на внешнем уровне, то переменная видима в пределах остатка исходного файла от места объявления ссылки.

В объявлениях, которые используют спецификатор класса памяти extern, инициализация не допускается, так как они ссылаются на переменные, значения которых уже определены.

Переменная, на которую делается ссылка extern, должна быть определена на внешнем уровне только один раз. Определение может быть сделано в любом из исходных файлов, составляющих программу.

6.4. ОБЪЯВЛЕНИЕ ФУНКЦИИ

Функции всегда определяются на внешнем уровне. Они могут объявляться на внешнем или внутреннем уровне со спецификатором класса памяти static или extern. В данной реализации объявление функции на внутреннем уровне имеет тот же смысл, что и объявление на внешнем уровне.

Правила видимости для функций отличаются от правил видимости для переменных и состоят в следующем.

1. Функция, объявленная как static, видима только в пределах исходного файла, в котором она определена. Любая функция в том же исходном файле может вызвать функцию с классом памяти static, а функции из других файлов — нет. Функция с классом памяти static и

с тем же именем может быть определена в другом исходном файле, и это не приведет к конфликту.

2. Функции, объявленные как `extern`, видимы в пределах всех исходных файлов, составляющих программу. Любая функция может вызывать функцию с классом `extern`.

3. Если в объявлении функции отсутствует спецификатор класса памяти, то функция по умолчанию считается с классом `extern`.

Если исходный файл содержит определение или объявление внешней переменной или функции, компилятор генерирует в объектном файле внешние имена и ссылки, которые разрешаются редактором связей при создании исполняемой программы. Часть внешних ссылок порождается из обращений к функциям библиотеки Си, поэтому для разрешения ссылок редактору связей должна быть доступна соответствующая библиотека функций.

6.5. ПРАВИЛА ОПРЕДЕЛЕНИЯ ВРЕМЕНИ ЖИЗНИ И ОБЛАСТИ ВИДИМОСТИ ОБЪЕКТА

Время жизни переменной (глобальное или локальное) определяется по следующим правилам.

1. Переменные, объявленные на внешнем уровне (т.е. вне всех блоков программы), всегда имеют глобальное время жизни.

2. Переменные, объявленные на внутреннем уровне (т.е. внутри блока), обычно имеют локальное время жизни. Однако можно обеспечить глобальное время жизни для переменной внутри блока, задавая класс памяти `static` при ее объявлении. Однажды объявленная с классом памяти `static` переменная будет сохранять свое значение на протяжении всего выполнения программы. Однако она будет "видима" только в блоке, в котором она определена, и во всех вложенных в него блоках.

Видимость переменных и функций в программе определяется следующими правилами.

1. Переменные, объявленные или определенные на внешнем уровне (т.е. вне всех блоков программы), видимы от точки объявления или определения до конца исходного файла. Можно объявить переменные видимыми и в других исходных файлах. Однако переменные, объявленные на внешнем уровне со спецификатором класса памяти `static`, видимы только в файле, в котором они определены.

2. Переменные, объявленные или определенные на внутреннем уровне (т.е. внутри блока), видимы, в общем случае, только от точки объявления или определения до конца блока, содержащего определение или объявление. Эти переменные называются локальными.

3. Переменные из объемлющих блоков, включая переменные, объявленные на внешнем уровне, видимы во внутренних блоках. Эту ви-

димось называют вложенной. Если переменная, объявленная внутри блока, имеет то же имя, что и переменная, объявленная на объемлющем уровне, то определение переменной в блоке заменяет определение объемлющего уровня на протяжении всего блока. Видимость переменной объемлющего уровня восстанавливается при выходе из блока.

4. Функции с классом памяти `static` видимы только в исходном файле, в котором они определены. Все другие функции являются глобально видимыми.

Метки в функции видимы на протяжении всей функции.

Имена формальных параметров, объявленные в списке параметров прототипа функции, видимы только от точки объявления параметра до конца объявления функции.

6.6. ПРАВИЛА ИНИЦИАЛИЗАЦИИ ДЛЯ ВНЕШНИХ И ВНУТРЕННИХ ПЕРЕМЕННЫХ

При инициализации необходимо придерживаться следующих правил.

1. Объявления, которые используют спецификатор класса памяти `extern`, не могут содержать инициаторов.

2. Переменные, объявленные на внешнем уровне, могут быть инициализированы. Если они явно не инициализированы, то они по умолчанию устанавливаются в нуль.

3. Любая переменная, объявленная со спецификатором класса памяти `static`, может быть инициализирована константным выражением. Инициализация переменных класса `static` производится в начале выполнения программы. Если отсутствует явная инициализация, то переменные класса памяти `static` по умолчанию устанавливаются в нуль.

4. Инициализация переменных со спецификаторами класса памяти `auto` и `register` выполняется всякий раз при входе в блок, в котором они объявлены. Если инициализация отсутствует в объявлении переменной `auto` или `register`, то начальное значение переменной не определено.

5. Начальными значениями для внешних объявлений переменной и для всех статических переменных, как внешних, так и внутренних, должны быть константные выражения. Так как адрес любой внешне объявленной или статической переменной является константой, то его можно использовать для инициализации указателя, объявленного на внешнем уровне с классом памяти `static`. Однако адрес автоматической переменной не может быть использован в качестве начального значения, так как он может меняться при каждом входе в блок.

Автоматические и регистровые переменные могут быть инициализированы константными выражениями или значениями переменных.

Пример.

```
int global_var;
int func (void)
{
    int local_var;
    static int *local_ptr = &local_var; /* Неправильное объявление */
    static int *global_ptr = &global_var; /* Правильное объявление */
    register int *reg_ptr = &local_var; /* Правильное объявление */
}
```

Глобальная переменная `global_var` объявляется на внешнем уровне, поэтому она имеет глобальное время жизни. Локальная переменная `local_var` имеет класс памяти `auto` и получает память только во время выполнения функции, в которой она объявлена. Поэтому нельзя инициализировать статический указатель `local_ptr` адресом переменной `local_var`. Статический указатель `global_ptr` может быть инициализирован адресом переменной `global_var`, так как этот адрес всегда один и тот же. Указатель `reg_ptr` является локальной переменной, и поэтому он может быть инициализирован неконстантным выражением.

6.7. КЛАССЫ ИМЕН

В соответствии с правилами видимости в языке Си допускается использование одного и того же идентификатора для более чем одного программного объекта.

Чтобы различать идентификаторы для различного рода объектов, компилятор устанавливает так называемые "классы имен". Во избежание противоречий имена внутри одного класса должны быть уникальными, однако в различных классах могут появляться идентичные имена. Это означает, что можно использовать один и тот же идентификатор для двух или более различных объектов, если объекты принадлежат к разным классам имен. Однозначное разрешение ссылок компилятор осуществляет по контексту данного идентификатора в программе.

Пример.

```
struct фио {
    char фио[20];
    int таб_номер;
    char адрес;
} фио;
```

В этом примере имена типа структуры, элемента структуры и переменной относятся к трем различным классам имен, поэтому не будет противоречий между тремя объектами с именем "фио". Компилятор определит по контексту программы, как интерпретировать каждый из случаев. Например, когда имя "фио" появляется после ключевого слова `struct`, это означает имя типа структуры. Когда имя "фио" появля-

ется после знака операции выбора элемента (-> или .), то это означает, что имя относится к элементу структуры. В другом контексте имя "фио" является переменной структурного типа.

К одному классу имен относятся имена переменных, функций, формальных параметров, констант перечисления и имена типов, вводимые с помощью `typedef`. Эти имена не могут дублироваться в пределах одной области видимости. Однако имена переменных, формальных параметров, констант перечисления и типов `typedef` могут быть переопределены внутри вложенных блоков.

Другой класс имен образуют имена типов перечислений, структур и смесей. Каждое имя типа должно отличаться от имен других типов в той же области видимости.

Отдельный класс имен составляют имена элементов каждой структуры или смеси. Имя элемента должно быть уникально внутри структуры или смеси, но имена элементов могут совпадать с именами других объектов, включая имена элементов других структур и смесей.

Метки операторов образуют отдельный класс имен. В одной функции все метки должны различаться, но они могут совпадать с метками других функций.

Глава 7

ДИРЕКТИВЫ ПРЕПРОЦЕССОРА И ПРАГМЫ

Мощным средством, позволяющим облегчить написание и отладку Си-программы, а также повысить ее читаемость, является препроцессор Си, или текстовый процессор, используемый для обработки текста исходного файла на первой фазе компилятора. Управление работой препроцессора производится посредством директив препроцессора, которые инструктируют препроцессор о выполнении определенных действий, например, замене лексемы в тексте, вставке содержимого внешнего файла в исходный файл, вставке группы операторов исходного файла, выбираемой в зависимости от некоторого условия. Обработанный препроцессором исходный текст поступает компилятору; он уже не содержит директив препроцессора.

Директивы препроцессора отмечаются специальным маркером — знаком номера (`#`). Знак номера (`#`) должен быть первым непробельным символом в строке, содержащей директиву. Между знаком номера и первой буквой директивы могут находиться пробелы.

Мы уже использовали в примерах программы директивы `#include` для включения в исходный файл других файлов и `#define` для опреде-

ления символических констант и макроопределений. Другие директивы препроцессора — это директива `#undef`, противоположная директиве `#define` и отменяющая определение идентификатора, директивы условной компиляции `#if`, `#else`, `#elif`, `#endif`, директива `#line`, позволяющая изменить нумерацию строк исходного текста, и директива `#error`, обеспечивающая возможность выдать сообщение во время компиляции в стандартный поток для вывода сообщений об ошибках и завершить компиляцию.

Директивы препроцессора могут появиться в любом месте исходного файла, но они применимы только к остатку исходного файла.

7.1. ДИРЕКТИВА `#DEFINE`

Директива `#define` имеет два формата.

Формат 1:

`#define идентификатор текст-подстановки`

Формат 2:

`#define идентификатор(список-параметров)
текст-подстановки`

Формат 1 определяет простую подстановку и используется для того, чтобы связать *идентификатор* с константой, ключевыми словами, часто используемыми операторами и выражениями. Идентификаторы, которые представляют константы, называются *символическими константами*, а идентификаторы, представляющие операторы или выражения, называются *макрокомандами*. В тексте исходного файла, следующем за директивой `#define`, все вхождения идентификатора заменяются на *текст-подстановки*, определённый в директиве справа от идентификатора. Идентификатор заменяется, если он образует лексему. Идентификатор не заменяется, если он является частью строки или другого идентификатора.

Текст подстановки представляет собой последовательность лексем, таких как ключевые слова и константы, или целых операторов. Один или более пробельных символов могут отделять текст от идентификатора. Эти пробельные символы не считаются частью текста подстановки, так же как и пробельные символы, следующие за последней лексемой текста. Если текст занимает больше, чем одну строку, то его можно продолжить на следующей строке, помещая символ обратной дробной черты перед символом "новая строка".

Текст подстановки может быть опущен. В этом случае удаляются все вхождения идентификатора из текста исходной программы. Тем не менее, идентификатор считается определённым и принимает значение 1, если проверяется директивой `#if`.

Определение символических констант с помощью директив `#define`

полезно в нескольких аспектах. Например, можно ввести короткое имя для длинной константы:

```
#define ПИ 3.1415926
```

Имеется соблазн определить ПИ следующей директивой:

```
#define ПИ = 3.1415926 /* это неверно! */
```

Однако смысл такой директивы `#define` в том, что вместо идентификатора ПИ будет подставляться текст `" = 3.1415926"`, что, очевидно, не то, что нужно. Препроцессор не выполняет проверки синтаксической правильности текста подстановки. Ошибки от неправильно определенного текста подстановки будут обнаружены только во время компиляции.

Другим полезным применением символических констант является определение констант, которые могут меняться в процессе эксплуатации программы, в одном месте, даже в отдельном файле. Если нужно константу поменять, изменение потребуется сделать только в одной директиве `#define`, связывающей имя константы с ее значением. Этот же прием позволяет упростить перенос программы с одной машины на другую, возможно с другой архитектурой.

Так, поставляемый с системой программирования файл включения `limits.h` содержит определения диапазонов значений основных типов данных:

```
#define CHAR_MAX 127
#define CHAR_MIN -128
#define UCHAR_MAX 255
#define SHRT_MAX 32767
#define SHRT_MIN -32768
#define USHRT_MAX 65535
#define INT_MAX 32767
#define INT_MIN -32768
#define UINT_MAX 65535
#define LONG_MAX 2147483647
#define LONG_MIN -2147483648
#define ULONG_MAX 4294967295
```

При переходе на машину с другими длинами типов данных следует изменить определения этих констант.

Полезно вводить mnemonicские имена для некоторых выражений или знаков операций. Например, у начинающих программистов на языке Си типичной ошибкой является употребление знака равно (`=`) в качестве знака операции сравнения, которая изображается двумя знаками равно (`==`). Можно ввести такое определение:

```
#define РАВНО ==
```

а также более привычные изображения знаков логических операций:

```
#define И &&
#define ИЛИ ||
```


Формат 2 директивы `#define` позволяет определить более сложные подстановки, а именно определить макрокоманды с аргументами.

Если после *идентификатора* следует *список-параметров*, то в исходном тексте, следующем за директивой `#define`, каждое вхождение выражения *идентификатор(список-аргументов)* заменяется на *текст-подстановки* с заменой формальных параметров фактически-ми аргументами.

Когда в директиве `#define` задан список параметров, то он содержит один или более формальных параметров, разделенных запятыми. Каждое имя в списке должно быть уникальным, и список должен быть заключен в круглые скобки. Не допускаются пробелы между идентификатором и открывающей скобкой. Область видимости формального параметра распространяется на все строки текста подстановки.

Имена формальных параметров в тексте подстановки отмечают места, куда должны быть подставлены фактические значения. Каждое имя формального параметра может встретиться в тексте более одного раза в любом порядке.

Фактические аргументы, следующие за *идентификатором* в исходном файле, сопоставляются с соответствующими формальными параметрами в списке параметров. Число фактических аргументов в списке аргументов должно совпадать с числом параметров в списке параметров. Каждый формальный параметр в тексте подстановки заменяется соответствующим фактическим аргументом. Если фактический аргумент содержит имя макроопределения (макрокоманду), то прежде чем фактический аргумент заменит соответствующий формальный параметр, заменится эта макрокоманда.

Если имя текущего макроопределения встретится в тексте подстановки (даже как результат другого макрорасширения), оно не замещается.

Эффективными макроопределениями являются определения функций `MIN`, `MAX`, `ABS` с помощью директивы `#define`:

```
/* Минимальное из значений x и y */
#define MIN(x,y) ((x)<(y)? (x):(y))
/* Максимальное из значений x и y */
#define MAX(x,y) ((x)<(y)? (y):(x))
/* Абсолютное значение x */
#define ABS(x) ((x)<0? -(x):(x))
```

После такого определения появление, например, макрокоманды `MAX(a,b)`

в тексте исходного файла в контексте

```
if (d < MAX(a,b))
    printf ("меньше");
```

приведет к подстановке

```
if (d < ((a)<(b)? (b):(a)))
    printf ("меньше");
```


Таким образом, обработка макроопределения не приводит к вызову функции со связанными с ним накладными расходами по передаче аргументов, распределению памяти, возврату значения, что имеет место, когда мы имеем дело с функциями. Макрокоманда обрабатывается на этапе препроцессирования, а не во время выполнения, и поэтому макрокоманды могут сократить время выполнения программы (хотя длина программы может увеличиться). Другим преимуществом макрокоманд является то, что они работают с любыми типами аргументов, в то время как для функции нужно или выполнять приведение типов аргументов или иметь несколько вариантов функций для разных типов аргументов.

Следует обратить внимание на две синтаксические особенности приведенных выше макроопределений: во-первых, параметры макроопределения заключены в круглые скобки; во-вторых, макроопределение не заканчивается точкой с запятой.

Точка с запятой, поставленная в конце макроопределения, появится и в макрорасширении, что породит синтаксически неправильную конструкцию. Отсутствие круглых скобок может привести к неправильному порядку вычисления, как, например, в случае определения

```
#define КУБ(x) (x*x*x)
```

и макрокоманды

```
КУБ(a+3),
```

которая заменится на

```
(a+3*a+3*a+3),
```

что, конечно, не является значением $(a+3)^3$.

Следует также помнить о возможных побочных эффектах при обработке макроопределений. Аргументы с побочным эффектом могут стать причиной непредсказуемых результатов, когда макроопределение содержит более одного вхождения данного формального параметра, которое будет вычисляться более чем один раз.

7.2. ПРЕПРОЦЕССОРНЫЕ ОПЕРАЦИИ

Если макроопределение имеет формальные параметры, над формальными параметрами в тексте подстановки при обработке макроопределения могут быть произведены препроцессорные операции: операция образования строки, которая обозначается одним знаком номера (#), и операция объединения лексем, которая обозначается двумя знаками номера (##).

Если знак операции образования строки (#) предшествует формальному параметру в макроопределении, то соответствующий фак-

тический аргумент, передаваемый макрокомандой, будет заключаться в кавычки и считаться строковым литералом.

Все пробелы, предшествующие первой лексеме и следующие за последней лексемой фактического аргумента, игнорируются. Все пробелы между лексемами фактического аргумента заменяются на один пробел в результирующем строковом литерале. Таким образом, если между двумя лексемами в фактическом аргументе встретится комментарий, то он заместится одним пробелом. Строковый литерал, получающийся в результате выполнения операции образования строки, автоматически сцепляется с соседними литералами, от которых он отделяется только одним пробелом. Более того, если символ, содержащийся в фактическом аргументе, при использовании в строковом литерале обычно задается управляющей последовательностью (например, символ кавычек " или обратной дробной черты \), то перед таким символом автоматически вставляется обратная дробная черта.

Пример.

```
#define строка(x) printf (#x "\n")
main()
{   строка (Строка - аргумент функции printf в кавычках);
    строка ("Строка будет в кавычках при выводе на экран");
    строка (Строка с управляющей последовательностью \");
}
```

При обработке этих макрокоманд будет произведена следующая подстановка:

```
printf ("Строка-аргумент функции printf в кавычках\n");
printf ("\\" Строка будет в кавычках при выводе на экран\" \n");
printf ("Строка с управляющей последовательностью \\\" \n");
```

При выполнении программы будет выдано:

```
Строка-аргумент функции printf в кавычках
"Строка будет в кавычках при выводе на экран"
Строка с управляющей последовательностью \"
```

Знак операции объединения лексем (##) позволяет объединить отдельные лексемы в одну. Поэтому знак операции не может задаваться перед первой лексемой и после последней лексемы в макроопределении.

Если формальный параметр в макроопределении предшествует знаку операции объединения лексем или следует за ним, то он сначала заменится фактическим аргументом. Если аргумент содержит макрокоманду, то она не обрабатывается. Затем лексемы, предшествующие знаку операции объединения или следующие за ним, сцепляются, а сам знак операции объединения удаляется из текста-подстановки. Результирующая лексема должна быть правильной лексемой. Если получившаяся в результате объединения лексема является именем макроопределения, то выполняется обработка макрокоманды.

Пример.

```
#define отладка(n) printf ("поле" #n "=%d", поле ##n)
```

Если задана макрокоманда

```
отладка(5);
```

она будет заменена на

```
printf ("поле5=%d", поле5);
```

Этот пример иллюстрирует использование операций образования строки и объединения лексем.

7.3. ДИРЕКТИВА #UNDEF

Директива `#undef` имеет формат:

```
#undef идентификатор
```

Директива `#undef` отменяет текущее определение *идентификатора*, заданного в директиве. Следовательно, все последующие вхождения идентификатора будут игнорироваться препроцессором. Для того чтобы отменить макроопределение с параметрами, достаточно задать в директиве `#undef` его идентификатор, не задавая список параметров.

Директива `#undef` может быть применена к идентификатору, который ранее не был определен. Это дополнительная гарантия того, что идентификатор не определен.

Директива `#undef` обычно используется совместно с директивой `#define` для создания области в исходной программе, в которой идентификатор имеет определенный смысл. Директива `#undef` также используется с директивой `#if` для управления условной компиляцией исходной программы.

После отмены идентификатор может быть сопоставлен с другим значением.

Пусть, например, мы хотим иметь локально видимый в одной функции нулевой указатель на `char`, а в другой — нулевой указатель на `double`. Можно определить `NIL` следующим образом:

```
#define NIL (char *)0
```

В конце функции отменить это определение:

```
#undef NIL
```

и ввести новое определение:

```
#define NIL (double *)0
```

7.4. ДИРЕКТИВА #INCLUDE

Директива `#include` имеет формат:

```
#include "спецификатор-маршрута"
```

```
#include <спецификатор-маршрута>
```


Директива `#include` добавляет содержимое заданного файла включения к другому файлу.

Директива `#include` предписывает препроцессору трактовать содержимое названного файла так, как если бы этот файл появился в исходной программе в точке, где записана директива. Включаемые файлы также могут содержать директивы препроцессора. Препроцессор выполняет директивы из нового текста, а затем продолжает обработку первоначального текста исходного файла.

Спецификатор-маршрута — это имя файла с предшествующей спецификацией каталога.

Препроцессор прекращает поиск, как только будет найден файл с данным именем. Если задан полный однозначный спецификатор маршрута, заключенный в кавычки " " или в угловые скобки <>, то препроцессор ищет файл по заданному маршруту и игнорирует стандартные каталоги. Стандартные каталоги — это каталоги, указанные в переменной операционной среды INCLUDE.

Если спецификатор маршрута не полный и заключен в кавычки, то препроцессор начинает поиск в каталоге, содержащем родительский файл. Родительский файл — это файл, содержащий директиву `#include`. Например, если файл с именем `wios.h` включается в файл `program`, то `program` является родительским файлом.

Включаемые файлы могут быть вложенными, т.е. директива `#include` может быть задана в файле, включаемом другой директивой `#include`. Например, если продолжить пример с файлами `program` и `wios.h`, предположив, что `wios.h` содержит директиву `#include` для файла `computer.h`, то `program` будет родительским файлом для файла `wios.h` и прародительским — для `computer.h`.

Если включаемые файлы являются вложенными, то поиск начинается с каталогов родительского файла и продолжается в каталогах прародительских файлов. Таким образом, поиск начинается с каталога, содержащего текущий исходный файл. Затем, если в данном каталоге файл не найден, поиск продолжается в каталогах, указанных в командной строке вызова компилятора и, наконец, в стандартных каталогах.

Если спецификация файла заключена в угловые скобки, то препроцессор не будет осуществлять поиск в текущем рабочем каталоге. Поиск будет начинаться с каталогов, указанных в командной строке вызова компилятора, а затем продолжается в стандартных каталогах.

Допускается до 10 уровней вложенности включаемых файлов.

Примером включаемого файла может быть совокупность определенных констант, знаков операций, макроопределений, приведенных в п. 7.1, которые полезны при разработке любой программы и могут быть добавлены к исходному файлу директивой `#include`.

Файлы включения полезны также для объявления внешних переменных и составных типов данных. С системой программирования Си поставляются готовые файлы включения, которые содержат определения символических констант, типов данных, используемых определенными категориями функций библиотеки Си, и прототипы этих функций. Эти файлы должны включаться в исходный файл, если функции используются в программе. Так, например, функции ввода-вывода потоком используют предопределенный тип структуры `FILE`, описывающий системную информацию о файле. Тип структуры `WINDOW` используется при программировании прикладных программ работы с окнами. Определения этих типов структур находятся соответственно в файлах включения `stdio.h` и `wios.h`. Готовые файлы включения облегчают программирование и сопровождение программ.

7.5. ДИРЕКТИВЫ УСЛОВНОЙ КОМПИЛЯЦИИ

Директивы условной компиляции — это директивы `#if`, `#elif`, `#else` и `#endif`, которые позволяют отменить компиляцию отдельных частей исходного файла посредством проверки константных выражений или идентификаторов, и имеют следующий формат:

```
#if ограниченное-константное-выражение
    текст
#elif ограниченное-константное-выражение
    текст
#elif ограниченное-константное-выражение
    текст
...
#else
    текст
#endif
```

Каждой директиве `#if` в исходном файле должна соответствовать закрывающая директива `#endif`. Между директивами `#if` и `#endif` могут использоваться директивы `#elif` и не более одной директивы `#else`. Директива `#else`, если она есть, должна быть последней директивой перед директивой `#endif`.

Препроцессор выбирает одно из вхождений *текста* в формате. Текст может занимать более одной строки. Обычно это участок программного текста, который имеет смысл для компилятора или препроцессора.

Выбранный текст обрабатывается препроцессором и передается на компиляцию. Если текст содержит директивы препроцессора, то эти директивы выполняются.

Вхождение текста, не выбранного препроцессором, удаляется из файла на этапе препроцессирования и поэтому не компилируется.

Препроцессор выбирает единственное вхождение текста на основе вычисления *константного-выражения*, следующего за каждой директивой `#if` или `#elif`, пока не будет найдено выражение со значением истина (не нуль). Выбирается текст, следующий за истинным выражением, до соответствующей директивы `#elif`, `#else` или `#endif`. Этот текст может содержать другие директивы препроцессора, начинающиеся со знака `#`.

Если все выражения ложны или отсутствуют директивы `#elif`, то препроцессор выбирает *текст* после директивы `#else`. Если к тому же отсутствует и директива `#else`, то никакой текст не выбирается.

Директивы `#if`, `#elif`, `#else` и `#endif` могут быть вложенными. Каждая вложенная директива `#else`, `#elif` или `#endif` принадлежит ближайшей предшествующей директиве `#if`.

Ограниченное-константное-выражение — это константное выражение, которое не содержит операций `sizeof`, приведения типов и констант перечисления. Ограниченное константное выражение, однако, может содержать препроцессорную операцию `defined`, которая имеет формат:

`defined(идентификатор)`

Константное выражение, содержащее операцию `defined`, считается истинным (не нулевым), если *идентификатор* в текущий момент определен, в противном случае — ложным. Даже если идентификатор определен директивой `#define` как пустой текст, он считается определенным.

Часто в качестве идентификатора в операции `defined` используются переменные, определяющие тип процессора или операционную систему и позволяющие выбрать код, подходящий для данной архитектуры процессора или учитывающий особенности операционной системы.

7.6. ДИРЕКТИВЫ `#LINE` И `#ERROR`

Директива `#line` имеет формат:

`#line константа ["имя-файла"]`

Директива `#line` предписывает компилятору изменение внутренней нумерации строк и имени файла на заданный *константой* номер и *имя-файла*. Компилятор использует номер строки и имя файла для ссылки на место ошибки, обнаруженной в процессе компиляции. Номер строки обычно соответствует номеру текущей входной строки, а именем файла обычно является имя текущего входного файла. Номер строки увеличивается после обработки каждой строки.

В случае изменения номера строки и имени файла компилятор игнорирует предыдущие их значения и продолжает обработку с новы-

ми значениями. Директива `#line` обычно используется генераторами для того, чтобы отнести сообщения об ошибках к первоначальному исходному файлу, а не к генерируемой программе.

Константа, заданная в директиве `#line`, может быть любой целой константой. *Имя-файла* может быть любой комбинацией символов, заключенной в кавычки. Если имя файла опущено, предыдущее имя файла сохраняется неизменным.

Текущие номер строки и имя файла доступны через предопределенные идентификаторы `__LINE__` и `__FILE__`. Идентификаторы `__LINE__` и `__FILE__` могут быть использованы при вставке в исходный файл программного текста для выдачи сообщений об ошибке.

Переменная `__FILE__` содержит строку, представляющую имя файла, заключенного в кавычки.

Проиллюстрируем использование идентификаторов `__LINE__` и `__FILE__` для указания места ошибки, которая выявляется посредством макрокоманды `ASSERT`. Аргументами макрокоманды `ASSERT` являются выражение, представляющее предположение программиста о поведении программы или утверждение, которое должно быть истинным, и текст сообщения, которое будет выводиться, если это предположение или утверждение оказывается ложным. Пусть макрокоманда `ASSERT` определена следующим образом:

```
#define ASSERT(условие, сообщение) \
    (!(условие)? printf("Утверждение ложно: %s \n", \
        (строка %d файла %s) \n", (сообщение), \
        __LINE__, __FILE__), 0 : 1)
```

Пусть параметром некоторой функции должен быть указатель на `char`. Тогда для проверки того, является ли указатель ненулевым, в теле функции можно использовать макрокоманду `ASSERT`:

```
func(char * строка)
...
if ASSERT(строка, "Нулевой указатель")
    return (0);
...
```

Если указатель окажется нулевым, будет выдано сообщение:

Утверждение ложно: Нулевой указатель (строка n файла "func.c")

Хотя это сообщение не избавляет от ошибки, оно помогает найти ошибку при отладке программы.

Директива `#error` имеет формат:

```
#error сообщение
```

Директива `#error` предписывает препроцессору выдать указанное *сообщение* в стандартный поток для вывода сообщений об ошибках `stderr` и завершить компиляцию.

7.7. ПРАГМЫ

В отличие от директив, которые являются указаниями для препроцессора и не попадают компилятору, прагмы — это инструкции компилятору языка Си, которые вставляются программистом в текст исходной программы на языке Си и используются для управления действиями компилятора в определенных частях программы, не влияя на программу в целом.

Формат прагмы:

`#pragma последовательность-символов`

Знак номера (#) должен быть первым отличным от пробела символом в строке, содержащей прагму. Между знаком номера и словом `pragma` могут быть пробелы.

С помощью прагм некоторые режимы компилятора можно установить не на всю Си-программу, а только на какую-то ее часть или на определенные функции.

Прагма `alloc_text` имеет формат:

`#pragma alloc_text (сегмент-текста, функция-1 [, функция-2] ...)`

Прагма управляет распределением *сегмента-текста*, в котором размещаются указанные *функции*, и должна задаваться после объявления перечисленных в ней функций, но до их определений. В одной единице компиляции может появиться не более 10 прагм `alloc_text`. Функции должны быть определены в том же модуле, что и прагма, и должны быть явно или неявно `near`.

Прагма `check_pointer` имеет формат:

`#pragma check_pointer ([on|off])`

Прагма устанавливает или запрещает контроль указателей для некоторых функций, оставляя режим умолчания для остальных. Надо поместить прагму `check_pointer` перед определением той функции, для которой надо установить (или запретить) контроль указателей. Прагма без аргумента возвращает контроль указателей к состоянию, определенному командной строкой.

Прагма `check_stack` имеет формат:

`#pragma check_stack ([on|off])`

Прагма устанавливает или запрещает контроль переполнения стека только для некоторых программ, оставляя режим умолчания для остальных. Надо поместить прагму `check_stack` перед определением той функции, для которой надо установить (или запретить) контроль переполнения стека. Прагма без аргумента возвращает контроль переполнения стека к состоянию, определенному командной строкой.

Прагма `function` имеет формат:

`#pragma function (функция-1 [, функция-2] ...)`

Прагма обеспечивает выборочную генерацию вызовов *функций* вместо встраиваемых функций. При компиляции программы должен быть установлен соответствующий режим генерации встраиваемых функций.

Прагма `intrinsic` имеет формат:

`#pragma intrinsic (функция-1 [,функция-2]...)`

Прагма генерирует встраиваемую форму для указанных *функций*.

Прагма `loop_opt` имеет формат:

`#pragma loop_opt ([on|off])`

Прагма подавляет или включает оптимизацию циклов для отдельных функций. Если прагма указана без аргументов, восстанавливается режим оптимизации циклов, указанный командной строкой.

Прагма `message` имеет формат:

`#pragma message (строка-сообщения)`

Прагма посылает *строку-сообщение* в `stdout` без прекращения компиляции.

Прагма `pack` имеет формат:

`#pragma pack ([1|2|4])`

Прагма указывает для определенных структур упаковку элементов, отличную от той, которая задана командной строкой. Чтобы восстановить режим упаковки, заданный командной строкой, надо указать прагму `pack` без аргументов.

Прагма `same_seg` имеет формат:

`#pragma same_seg (переменная-1 [,переменная-2]...)`

Все перечисленные внешние переменные распределяются в один и тот же сегмент данных.

Компилятор интегрированной среды воспринимает только прагмы `check_pointer`, `check_stack`, `message` и `pack`. Для оптимизирующего компилятора действительны все названные прагмы.

Часть 2

НЕКОТОРЫЕ РЕКОМЕНДАЦИИ ПО ПРОГРАММИРОВАНИЮ

Глава 8

УКАЗАТЕЛИ И АДРЕСНАЯ АРИФМЕТИКА

Язык Си поддерживает адресную арифметику, т.е. арифметику с использованием адресов памяти. Для представления адреса предназначен тип данных, называемый указателем. Указатель — это переменная, содержащая адрес другого объекта Си-программы. Если переменная содержит адрес некоторого другого объекта, то говорят, что переменная указывает на этот объект. Указатель может указывать на обычную переменную, массив, строку, структуру, другой указатель, функцию. В общем случае указатель может указывать не только на объект программы. Он может содержать адрес любого объекта из памяти компьютера, например, адрес переменной из области данных BIOS или адрес буфера видеомонитора.

Использование указателей позволяет создавать более быстрые и компактные программы, так как указатели позволяют ссылаться на объекты программы так же, как это делает машина. Применение указателей необходимо или полезно в следующих ситуациях:

- при динамическом распределении памяти;

- при работе с файлами;

- передача указателей в качестве параметров функции позволяет вызываемой функции изменять данные вызывающей функции;

- указатель является альтернативным (по отношению к индексам) способом работы с массивом и позволяет обеспечить передачу массива для обработки вызываемой функцией;

- можно определить указатель на функцию и использовать этот указатель для вызова функции. Это дает возможность передавать функции в функции.

8.1. ОПРЕДЕЛЕНИЕ УКАЗАТЕЛЕЙ

Объявление указателей рассматривалось в п. 2.4. Здесь будут описаны некоторые приемы программирования с использованием указателей.

Объявление

```
char *cptr;
```

означает, что объявляется указатель (с именем `cptr`) на объект типа

char. Этим объектом может быть переменная типа char (символьная переменная), массив, элементами которого являются символы (строка Си), или блок памяти, полученный, например, при динамическом распределении памяти.

Аналогично, указатель iptr, объявленный как

```
int *iptr;
```

может указывать на переменную целого типа, массив (одномерный или многомерный) из элементов целого типа или какой-то другой объект в основной памяти.

Каждое из этих объявлений выделяет память для переменной типа указатель, но каждый из указателей cptr и iptr пока ни на что не указывает. До тех пор, пока указателю не будет присвоен адрес какого-либо объекта, его нельзя использовать в программе.

Для получения адреса какого-либо объекта используется операция "адрес" (&). Например, выражение &ival, где ival переменная целого типа, означает "получить адрес переменной ival". Используя операцию "адрес", указателям можно присваивать адреса объектов. Например:

```
char cval, *cptr;  
int ival, *iptr;  
cptr = &cval;  
iptr = &ival;
```

Теперь указатель cptr содержит адрес переменной cval, а указатель iptr — адрес переменной ival.

Доступ к объекту через указатель осуществляется с помощью операции "разадресации" (*), которая рассматривает свой операнд как адрес некоторого объекта и использует этот адрес для выборки содержимого. Типом результата операции разадресации является тип объекта, адресуемого указателем.

Пример.

```
#include <stdio.h>  
main ()  
{  
    int ival, *iptr;  
    ival=50;  
    iptr=&ival;  
    printf("%d\n", *iptr); /* Печатается число 50 */  
}
```

Тип, адресуемый указателем, определяет количество байтов, извлекаемых из памяти при выполнении операции разадресации. В данном примере операция разадресации в операторе printf выполняется следующим образом: из памяти извлекается число байтов, равное размеру типа int (sizeof (int)), т.е. два байта. Эти два байта преобразуются к типу int и результат печатается.

Указатель всегда должен указывать на объект того же типа, с которым он объявлен. В противном случае, если указатель на объект одного типа содержит адрес объекта другого типа, то использование такого указателя приводит к бессмысленным результатам.

Пример.

```
#include <stdio.h>
main ()
{
    float f;
    int *pi;
    f=100.25;
    pi=&f;
    printf("%d\n", *pi);    /* Печатается число -32768 */
}
```

Указателю pi на тип int присваивается адрес переменной f типа float. При выполнении операции разадресации из памяти извлекаются младшие два байта переменной с плавающей точкой. Переменная f в памяти выглядит, как показано на рис. 8.

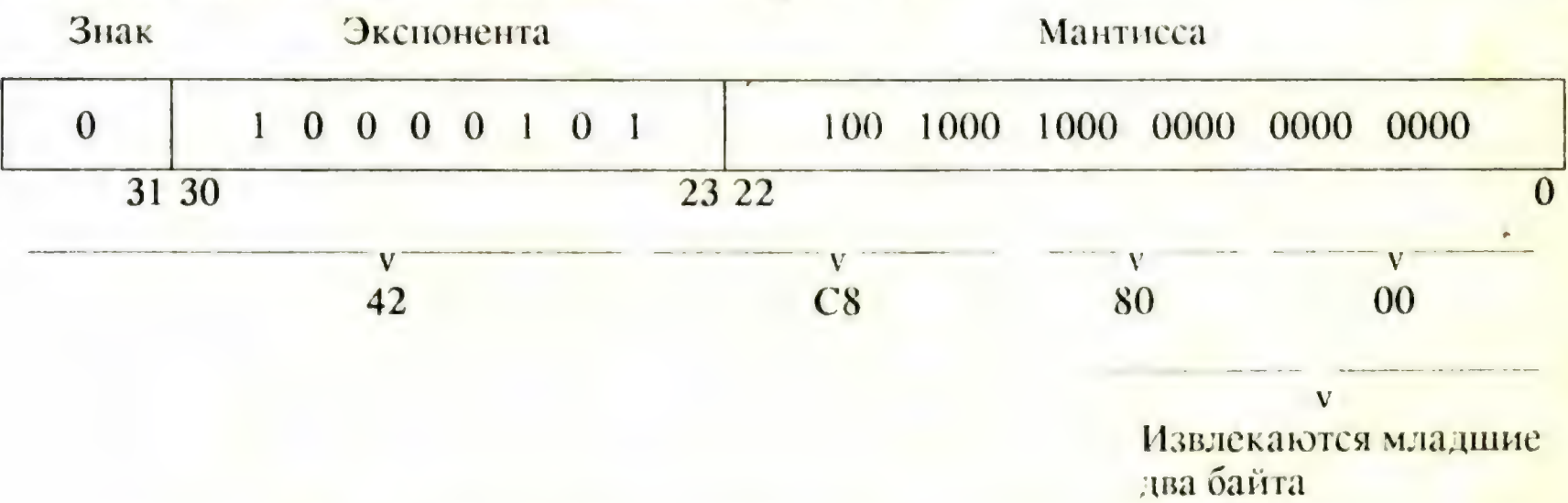


Рис. 8. Представление переменной с плавающей точкой

Младшие два байта переменной f (шестнадцатеричные 8000) преобразуются к типу int. Результатом является десятичное число -32768, которое и печатается.

8.2. УКАЗАТЕЛЬ НА НЕОПРЕДЕЛЕННЫЙ ТИП

Существует специальный тип указателя, называемый указателем на неопределенный тип. Для объявления такого указателя вместо спецификатора типа указателя задается ключевое слово void в следующем формате:

```
void * имя-указателя;
```

Ключевое слово void в объявлении указателя позволяет отсрочить определение типа, на который ссылается указатель. Указатель на тип void может быть использован для ссылки на объект любого типа. Но для того чтобы оперировать таким указателем или объектом, который он

адресует, необходимо явно задать требуемый тип в каждой операции с указателем. Это можно сделать с помощью операции приведения типа.

Пример.

```
/* Эта программа демонстрирует использование указателя на
   неопределенный тип */
#include <stdio.h>
main()
{
    int a = 125;
    double d = 10.39375;
    void *vp;
    /* Присвоить адрес целой переменной указателю vp */
    vp = &a;
    /* Привести указатель vp к типу int и разадресовать его */
    printf("a = %d\n", *((int *)vp));
    /* Присвоить адрес плавающей переменной указателю vp */
    vp = &d;
    /* Привести указатель vp к типу double и разадресовать его */
    printf("d = %lf\n", *((double *)vp));
}
```

Указатель `vp` объявлен как указатель, который может указывать на объект любого типа. Сначала указателю `vp` назначается адрес целой переменной `"a"`. Однако ни одна операция не может быть выполнена над указателем `vp` до тех пор, пока явно не будет задан тип, на который он указывает. Указатель `vp` преобразуется к указателю на тип `int` и выполняется операция разадресации. Печатается значение переменной `"a"`. Затем указателю `vp` присваивается адрес переменной `d` типа `double`. Чтобы получить доступ к переменной `d` через указатель `vp`, он должен быть преобразован к типу `double`.

8.3. ВЫРАЖЕНИЯ С УКАЗАТЕЛЯМИ

Присваивание указателей

Можно присваивать один указатель другому. Однако эта операция имеет смысл только в том случае, если оба указателя адресуют один и тот же тип.

Пример.

```
/* Эта программа демонстрирует присваивание указателей */
#include <stdio.h>
main()
{
    int i = 12525, *pi1, *pi2;
    pi1 = &i;
    pi2 = pi1;
    printf("%d\n", *pi1); /* Оба оператора напечатают одно */
    printf("%d\n", *pi2); /* и то же */
}
```


Сравнение указателей

В общем случае сравнение двух указателей одной из операций отношения имеет смысл в том случае, если оба указателя адресуют общий для них объект, например, массив или строку.

Пример.

```
/* Сравнение на неравенство указателей */
#include <stdio.h>
main()
{
    int a, b;
    int *p1, *p2;
    p1 = &a;
    p2 = &b;
    /* Указатели p1 и p2 не равны, так как они указывают на
       разные объекты */
    if (p1 != p2)
        printf("Указатели p1 и p2 не равны\n");
    else
        printf("Эта строка не напечатается\n");
}
```

Использование указателей в арифметических операциях

С указателями могут быть использованы только операции "увеличения" (++), "уменьшения" (--), "сложения" (+) и "вычитания" (-).

Если к указателю применяются операции ++ или --, то указатель увеличивается или уменьшается на размер объекта, который он адресует, что условно можно представить так:

```
тип * ptr;
ptr++ = содержимое ptr + sizeof (тип)
ptr-- = содержимое ptr - sizeof (тип)
```

где типом может быть любой основной тип языка Си или тип структуры или смеси.

Пусть имеется структура:

```
struct S {
    char c;
    int i;
    double d;
} s;

struct S *ps;
s.c = 'd';
s.i = 10;
s.d = 33.125;
ps = &s;
ps++;
```

Указатель ps увеличивается на размер адресуемой им структуры, представленной на рис. 9, по следующей формуле:

$ps++ = \text{содержимое } ps + \text{sizeof}(\text{struct } S) =$
 $= \text{содержимое } ps + 12 = 120 + 12 = 132$

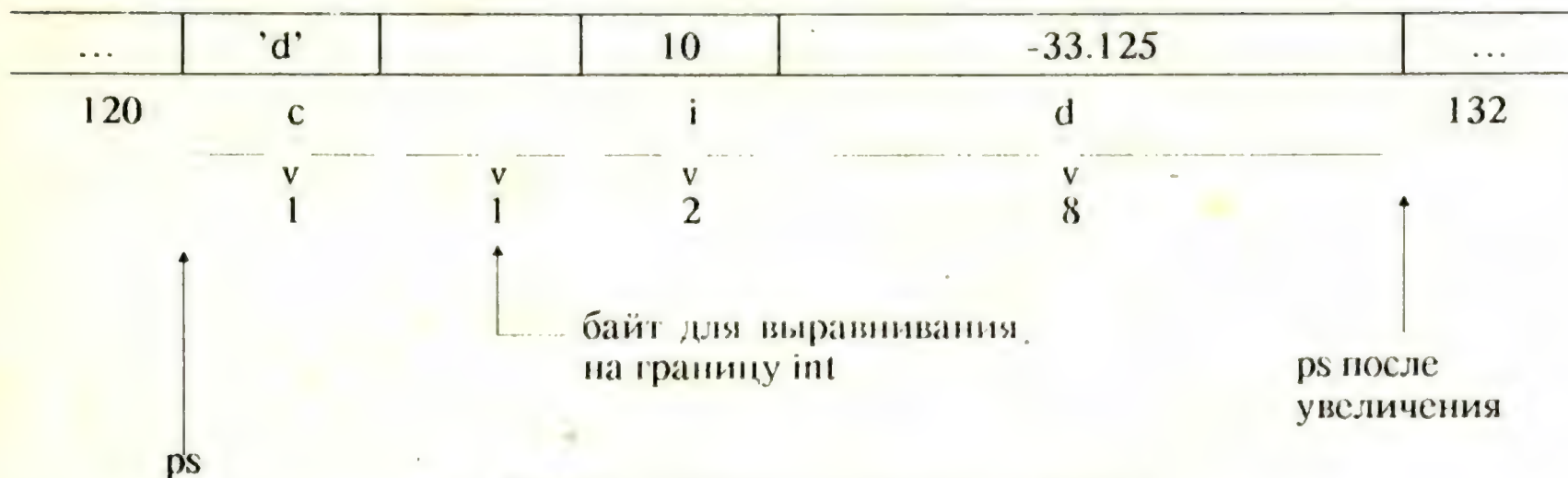


Рис. 9. Представление структуры в памяти

Одним из операндов операции сложения может быть указатель. Другим операндом должно быть целое число. Целое число складывается с указателем следующим образом:

```
тип *ptr;
int n;
ptr + n = содержимое ptr + n * sizeof(тип)
```

Левым операндом операции вычитания может быть указатель, а правым операндом должно быть целое число. Вычитание целого из указателя:

```
тип * ptr;
int n;
ptr - n = содержимое ptr - n * sizeof(тип)
```

К указателям можно применять только описанные операции. Следующие операции недопустимы с указателями:

- сложение двух указателей;
- вычитание двух указателей на различные объекты;
- сложение указателей с числом с плавающей точкой;
- вычитание из указателей числа с плавающей точкой;
- умножение указателей;
- деление указателей;
- поразрядные операции и операции сдвига указателей.

Пример.

```
/* Этот пример демонстрирует использование операций
 * с указателем. Вычисляется контрольная сумма последо-
 * вательности байтов, ограниченной двумя указателями.
 */
unsigned char checksum(unsigned char *begin, unsigned char *end)
{
    unsigned char chk_sum = 0x0;
    while (begin <= end)
        chk_sum += *begin++;
    chk_sum = ~chk_sum;
    chk_sum += 1;
    return(chk_sum); }
```


8.4. ПРЕОБРАЗОВАНИЕ УКАЗАТЕЛЕЙ

Указатель на объект одного типа может быть преобразован в указатель на другой тип. Однако при этом следует учитывать, что объект, адресуемый преобразованным указателем, будет интерпретироваться по-другому (может измениться его размер, а следовательно, и значение). Преобразование указателей выполняется операцией приведения типа указателя:

(тип *) указатель

Пример.

```
/* Преобразование указателей */
#include <stdio.h>
main()
{
    int i, *ptr;
    i = 0x8e41;
    ptr = &i;
    printf("%d\n", *ptr);           /* Печатается -29119 */
    printf("%d\n", *((char *) ptr)); /* Печатается 65 */
}
```

Объекты, адресуемые указателями `ptr` и `(char *) ptr`, выглядят, как показано на рис. 10.

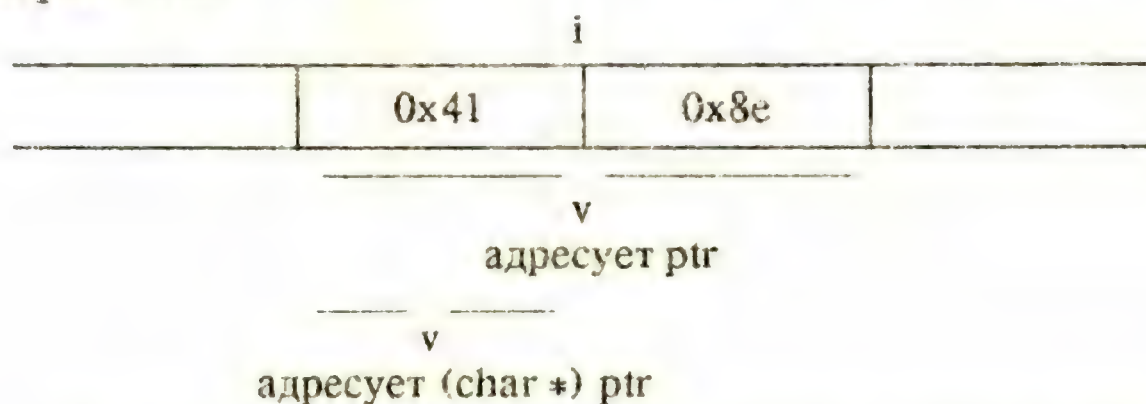


Рис. 10. Объекты, адресуемые различными указателями

8.5. СВЯЗЬ УКАЗАТЕЛЕЙ И МАССИВОВ

Между указателями и массивами существует прямая связь. Когда объявляется массив, например

```
int arr[25];
```

то этим определяется несколько больше, чем только выделение памяти для двадцати пяти элементов типа `int`. А именно, идентификатор массива `arr` определяется как константный указатель на первый (нулевой) элемент массива (адрес первого элемента массива `&arr[0]`). Так как идентификатор массива определяется как адрес, то следующее присваивание является допустимым:

```
int arr[25];
int *ptr;
ptr = arr;
```


Указатель `ptr` устанавливается на адрес первого элемента массива `arr`. Указатель `ptr` становится указателем на массив и его можно использовать для ссылки на элементы массива.

Присваивание `ptr = arr` можно записать в эквивалентной форме

```
ptr = &arr[0];
```

Для присваивания указателю `ptr` адреса начала массива `arr` можно использовать любую форму записи.

Для доступа к элементам массива тоже существуют два способа.

Первый способ доступа к элементам массива связан с использованием обычных индексных выражений вида `arr[i]`. Например, индексное выражение `arr[5]` ссылается на шестой элемент массива `arr`.

Второй способ доступа к элементам массива связан с использованием адресного выражения (или выражения с указателем на массив)

```
*(ptr + i)
```

Например, адресное выражение `*(ptr+5)` ссылается на шестой элемент массива `arr`.

В общем случае индексное выражение в языке Си имеет вид:

выражение-1 [выражение-2],

где выражение-1 — это любой указатель, например, идентификатор массива, а выражение-2 — это целая величина. Однако требуется только, чтобы одно из выражений было указателем, а второе — целой величиной. Поэтому выражение-1 может быть целой величиной, а выражение-2 — указателем.

Рассмотрим, как вычисляется индексное выражение `arr[5]`. Идентификатор `arr` является указателем начала массива. В соответствии с правилами выполнения операции сложения указателя с целой величиной целая величина преобразуется к адресному представлению путем умножения ее на размер типа, адресуемого указателем. Целая константа 5 умножается на размер типа `int`. Преобразованное значение числа 5 представляет 5 позиций типа `int`. Это преобразованное значение числа 5 складывается со значением указателя `arr`, что дает адрес, относящийся к шестой позиции типа `int` после `arr`. К полученному адресу применяется операция "разадресация". Результатом является значение шестого элемента массива. В терминах языка Си это выглядит так:

```
arr[5] = *(arr + 5) = *(arr + 5 * sizeof(int))
```

Следовательно, следующие выражения ссылаются на шестой элемент массива `arr`:

```
arr[5]  
5[arr]  
*(arr+5)  
*(5+arr)
```

Как следует из определения индексного выражения Си, любой указатель может быть задан с индексом. Указатель на массив также может быть проиндексирован. Выражения


```
ptr[5]
5[ptr]
```

ссылаются на тот же шестой элемент массива `arr`. Если расписать индексное выражение `ptr[5]`, то получится адресное выражение

```
*(ptr + 5)
```

В действительности компилятор Си преобразует все индексные выражения в адресные выражения, например

```
arr[i] в *(arr + i),
ptr[i] в *(ptr + i)
```

Таким образом, для доступа к i -му элементу массива `arr` можно пользоваться любым из данных выражений:

```
arr[i]      i[arr]      *(arr+i)
*(i+arr)    *(ptr+i)    *(i+ptr)
ptr[i]      i[ptr]
```

или, другими словами, любое индексное выражение можно представить в виде выражения с указателем и смещением и наоборот. Причем обе формы доступа к массиву можно сочетать в одном операторе.

Пример.

```
/*
 * Функция strcpy — копирование строк
 * Вариант с использованием массивов
 */
char * strcpy(char s1[], char s2[])
{
    int i;
    for (i=0; s1[i]=s2[i]; i++);
    return (s1);
}
/*
 * Функция strcpy — копирование строк
 * Вариант с использованием указателей
 */
char * strcpy(char *s1, char *s2)
{
    char *str = s1;
    while (*s1++ = *s2++);
    return (str);
}
```

Операции с указателями на массивы

Если два указателя `ptr1`, `ptr2` ссылаются на один и тот же массив

```
int arr[SIZE];
int *ptr1, *ptr2;
int i;
ptr1 = ptr2 = arr;
...
```

то к ним можно применить операции, перечисленные ниже.

Операции с указателями на массив

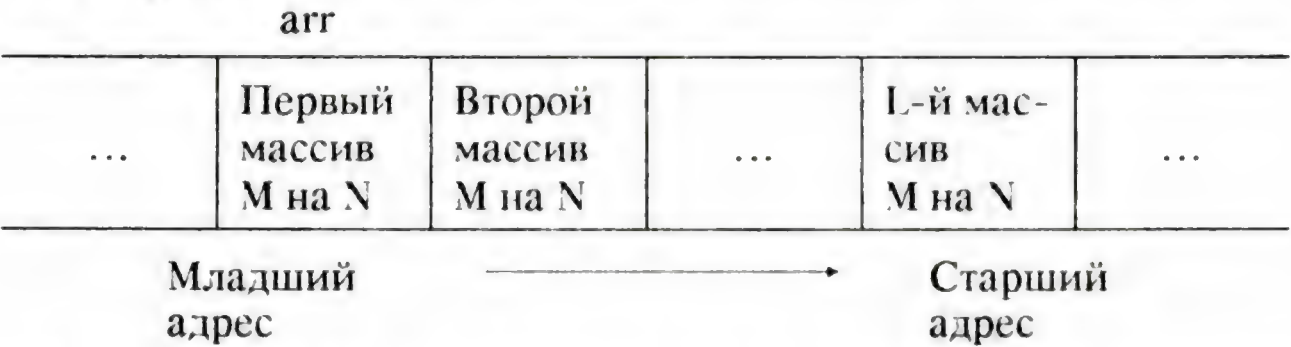
Операция	Результат операции
ptr1 ++ ptr1 --	Устанавливает ptr1 на следующий (предыдущий) элемент массива arr
ptr1 + i ptr1 - i	Устанавливает ptr1 на i-й элемент после (перед) ptr1
ptr2 - ptr1	Дает число элементов между ptr1 и ptr2 (отрицательное число, если ptr2 указывает на элемент с меньшим индексом)
ptr1 == ptr2	Результат "истина" (1), если ptr1 и ptr2 указывают на один и тот же элемент массива arr. В противном случае "ложь" (0)
ptr1 != ptr2	Результат "истина" (1), если ptr1 и ptr2 указывают на разные элементы массива arr. В противном случае "ложь" (0)
ptr1 < ptr2 ptr1 <= ptr2	Результат "истина" (1), если ptr1 указывает на элемент массива arr с меньшим (или равным) индексом, чем ptr2. В противном случае "ложь" (0)
ptr1 > ptr2 ptr1 >= ptr2	Результат "истина" (1), если ptr1 указывает на элемент массива arr с большим (или равным) индексом, чем ptr2. В противном случае "ложь" (0)

Указатели на многомерные массивы

Многомерный массив в языке Си — это массив массивов, т.е. массив, элементами которого являются массивы. Например, трехмерный массив есть массив, элементы которого двухмерные массивы

```
int arr[L][M][N];
```

Массив arr состоит из L элементов. Каждый элемент — это двухмерный массив размерностью M на N. В памяти массив arr расположен следующим образом:



Каждый массив M на N в памяти располагается по строкам. Например, i-й массив M на N трехмерного массива arr представляется следующим образом:

	Первый массив из N элементов	Второй массив из N элементов	...	M-й массив из N элементов	
--	------------------------------	------------------------------	-----	---------------------------	--

Рассмотрим, как осуществлять доступ к элементам многомерного массива через указатель. Сделаем это на примере массива `arr`.

Объявим указатель и свяжем его с массивом `arr`

```
int *ptr;
ptr = &arr[0][0][0];
```

Необходимо получить доступ к элементу `arr[i][j][k]`, или k -му элементу j -й строки i -го массива M на N массива `arr`. Последовательно это вычисляется так:

адрес первого массива M на N в массиве <code>arr</code>	<code>ptr</code>
адрес i -го массива M на N в массиве <code>arr</code>	<code>ptr+i*(M*N)</code>
адрес j -й строки i -го массива M на N в массиве <code>arr</code>	<code>ptr+i*(M*N)+j*N</code>
адрес k -го элемента j -й строки i -го массива M на N в массиве <code>arr</code>	<code>ptr+i*(M*N)+j*N+k</code>
k -й элемент j -й строки i -го массива M на N в массиве <code>arr</code>	<code>*(ptr+i*(M*N)+j*N+k)</code>

Пример.

```
/* Функция поиска минимального элемента в трехмерном
 * массиве. В функцию передается адрес начала массива и
 * размеры массива. Возвращается указатель на структуру,
 * содержащую индекс минимального элемента.
 */
```

```
struct INDEX { /* Определение структуры INDEX */
```

```
    int i ,
```

```
        j ,
```

```
        k;
```

```
    } index;
```

```
struct INDEX *find(int *ptr, int L, int M, int N)
```

```
{
```

```
    int min;
```

```
    int i, j, k, ind;
```

```
    min = *ptr;
```

```
    index.i = index.j = index.k = 0;
```

```
    for (i = 0; i < L; i++)
```

```
        for (j = 0; j < M; j++)
```

```
            for (k = 0; k < N; k++) {
```

```
                ind = i * (M * N) + j * N + k;
```

```
                if (min > *(ptr + ind)) {
```

```
                    min = *(ptr + ind);
```

```
                    index.i = i;
```

```
                    index.j = j;
```

```
                    index.k = k;
```

```
                }
```

```
            }
```

```
    return &index;
```

```
}
```


Глава 9

МОДЕЛИ ПАМЯТИ

Термин "модель памяти" для программы на языке Си означает то, как программа использует память процессора, и связан с его архитектурой.

9.1. НЕКОТОРЫЕ СВЕДЕНИЯ О ПРОЦЕССОРЕ

Процессор типа М86 имеет четырнадцать 16-битовых регистров:

Регистры общего назначения

AX	AH	AL	Регистр-аккумулятор
BX	BH	BL	Базовый регистр
CX	CH	CL	Регистр-счетчик
DX	DH	DL	Регистр данных

v

8 бит

v

8 бит

Индексные регистры и регистры указатели

SP		Указатель стека
BP		Дополнительный указатель стека
SI		Индексный регистр источника
DI		Индексный регистр назначения

Сегментные регистры

CS		Указатель сегмента кода
DS		Указатель сегмента данных
SS		Указатель сегмента стека
ES		Указатель дополнительного сегмента

Регистры специального назначения

IP		Указатель команд
		Регистр флажков

Регистры общего назначения используются как рабочие регистры. Каждый регистр можно рассматривать и как пару самостоятельных 8-битовых регистров. Кроме функций общего назначения, каждый регистр может использоваться и для специальных целей:

AX — в большинстве арифметических и логических операций;

BX — для хранения смещения указателя far;

CX — в командах повторения;

DX — для хранения данных.

Индексные регистры и регистры указатели используются в относительной адресации, как указатели стека и в командах пересылки блоков данных. Регистр SP является указателем вершины стека (смещением в сегменте стека).

Регистр BP является дополнительным указателем стека. Компиляторы системы программирования Си используют регистр BP как базовый адрес для аргументов и локальных переменных функции Си, которые размещаются компилятором в стеке. Аргументы, размещаемые в стеке, имеют положительное смещение от базового адреса в регистре BP (представление смещения зависит от модели памяти). Локальные переменные имеют отрицательные смещения от базового адреса в регистре BP. Сам регистр BP всегда указывает на сохраненное в стеке предыдущее значение регистра BP.

Регистры SI и DI могут использоваться как регистры общего назначения и как индексные регистры. Кроме того, компиляторы Си используют эти регистры для переменных с классом памяти register (поэтому регистровая память гарантируется только для первых двух локальных переменных).

Сегментные регистры процессора M86 предназначены для поддержки сегментной организации памяти и содержат начальные адреса четырех сегментов программы.

Регистры специального назначения включают регистр IP, который является указателем следующей выполняемой процессором команды в сегменте кода, и регистр флажков, который содержит информацию о текущем состоянии процессора:

9.2. ОРГАНИЗАЦИЯ ПАМЯТИ И АДРЕСАЦИЯ

Процессор M86 является процессором с сегментной организацией памяти. Он позволяет адресовать 1 Мбайт памяти. Так как все регистры-указатели процессора M86 (BX, SP, BP, SI, DI и IP) имеют 16 бит, то прямой доступ имеют только 64 Кбайта памяти. Эти 64 Кбайта памяти называются сегментом. Для того чтобы адресовать 1 Мбайт памяти, требуется 20-битовый адрес. Поэтому для представления 20-битового адреса используется два регистра (32 бита). Один регистр содержит адрес сегмента, который задается одним из сегментных регистров (CS, DS, SS или ES). Второй регистр является смещением в сегменте. Процессор использует четыре сегмента: кода, данных, стека и дополнительный. Сегмент кода содержит код программы, сегмент данных — данные программы, сегмент стека — стек программы, а

дополнительный сегмент является дополнительной рабочей памятью программы. Полный 20-битовый адрес, адресующий все адресное пространство процессора, вычисляется следующим образом: значение сегментного регистра сдвигается влево на четыре бита (на одну шестнадцатеричную цифру) и к полученному значению добавляется смещение. Например, если сегментный регистр содержит значение 10FA, а смещение равно 1C2, то фактический адрес памяти получается по следующей схеме:

Сегментный регистр:	0001	0000	1111	1010	=	10fa
Сегментный регистр после сдвига:	0001	0000	1111	1010	0000	= 10fa0
Смещение:		0000	0001	1100	0010	= 01c2
Адрес	0001	0001	0001	0110	0010	= 11162

Как видно из примера, начальный адрес сегмента всегда является 20-битовым числом, а так как сегментный регистр имеет только 16 бит, недостающие младшие 4 бита всегда подразумеваются равными нулю. Это означает, что сегменты всегда начинаются на границе 16 байт или параграфа (отрезок памяти 16 байт называется параграфом).

Адрес памяти представляется в форме "сегмент:смещение", поэтому рассмотренный в примере адрес задается в виде 10FA:01C2.

Сегменты могут быть смежными, разделенными, перекрываться полностью или частично. Так как сегменты могут перекрываться, то одна и та же ячейка памяти может содержаться во многих сегментах. Это означает, что она может быть адресована более чем одним адресом. Например, следующие адреса указывают на один и тот же байт памяти:

- 10FA:01C2
- 10F0:0262
- 10E0:0362
- 10A0:0762

9.3. СТРУКТУРА ОБЪЕКТНОЙ СИ-ПРОГРАММЫ

Когда Си-программа, откомпилированная одним из компиляторов системы программирования Си, загружается в основную память, код и данные программы загружаются в сегменты памяти. Для кода и данных программы распределяются отдельные сегменты. Это означает, что минимальное число сегментов, распределяемых для программы, равно двум. Эти два сегмента, требуемые для Си-программы, называются сегментами по умолчанию.

Общая структура объектной Си-программы показана на рис. 11.

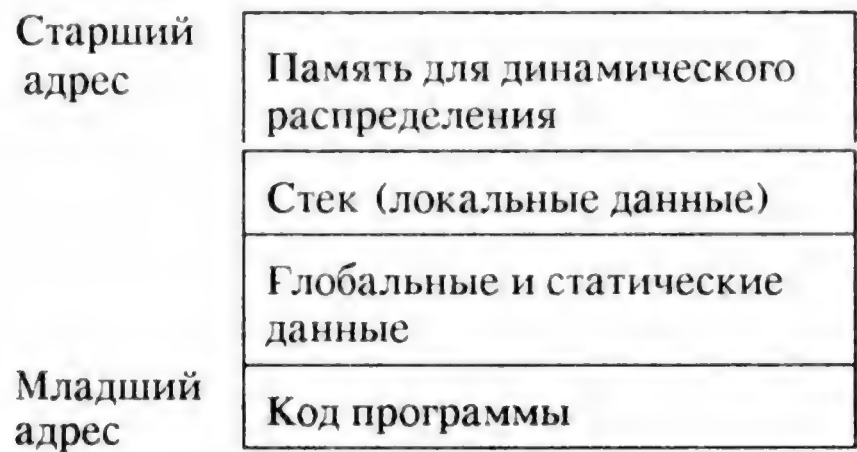


Рис. 11. Структура объектной Си-программы

Задание различных моделей памяти для программы будет вызывать некоторые изменения в организации памяти программы, но общая схема будет оставаться той же.

9.4. АДРЕСАЦИЯ NEAR, FAR И HUGE

Специальные ключевые слова `near`, `far` и `huge` используются в Си-программе для модификации объявлений переменных и функций. Они определяют отличное от стандартного размещение переменных и функций в объектной Си-программе.

Когда эти ключевые слова используются с указателями, то они изменяют размер указателя, который определяется выбранной моделью памяти. Ключевые слова `near`, `far` и `huge` определяют три типа указателей (три типа адресации): `near` (16 бит), `far` (32 бита) и `huge` (32 бита).

Адрес `near`

Доступ внутри сегмента по умолчанию возможен через 16-битовое смещение, так как адрес сегмента, распределяемого для сегмента по умолчанию, всегда известен. Например, адрес данного в сегменте данных по умолчанию получается сложением 16-битовой величины указателя на данное с содержимым регистра сегмента данных (DS), сдвинутым влево на четыре бита. Это 16-битовое смещение называется адресом `near`. Доступ к данным из сегмента по умолчанию в Си-программах осуществляется через указатели `near`, объявление которых выглядит как:

```
тип near *near_pointer;
```

Так как для доступа к данным через адрес `near` требуется только 16-битовая арифметика, то ссылки `near` наиболее эффективны.

Адрес far

Когда данные выходят за пределы сегмента по умолчанию, адрес должен состоять из двух частей: адреса сегмента и смещения. Такие адреса называются адресами far. Доступ к данным вне сегмента по умолчанию в Си-программах осуществляется через указатели far, которые объявляются следующим образом:

```
тип far *far_pointer;
```

Указатели far позволяют адресовать всю память. Однако в использовании указателей far имеется несколько нюансов.

Пусть имеется три указателя far ptr1, ptr2 и ptr3 и они задают три пары значений, соответственно:

```
ptr1 — 5f20:0210,  
ptr2 — 5f21:0200,  
ptr3 — 5f41:0000.
```

Так как над указателями допустимы операции сравнения, то правомерны следующие выражения:

```
ptr1 == ptr2  
ptr1 == ptr3  
ptr2 == ptr3
```

Однако результатом во всех трех выражениях будет "ложь", так как операции == и != над указателями far используют все 32 бита указателя как длинное целое без знака (unsigned long), а не как фактический адрес памяти.

С другой стороны, операции <, <=, > и >= при сравнении указателей far используют только 16 бит смещения (как целое без знака). Поэтому для указателей far также не гарантируется правильность выполнения этих операций. Например, вычисление выражений

```
ptr1 > ptr2  
ptr1 > ptr3  
ptr2 > ptr3
```

приводит к неожиданному результату: значением выражений будет "истина", хотя в действительности все три указателя адресуют одну и ту же ячейку памяти.

Когда указатель far увеличивается или уменьшается, то изменяется только смещение, и если изменение смещения приводит к превышению максимально допустимого значения (FFFF) либо к уменьшению минимально допустимого значения (0000), то указатель будет устанавливаться на начало либо конец сегмента. Например, если добавить 1 к указателю 1000:ffff, то результатом будет 1000:0000, а не 2000:0000. Аналогично, если вычесть 1 из указателя 1000:0000, то результатом будет 1000:ffff, а не 0fff:000f. Это означает, что указате-

лем far нельзя адресовать данные, размер которых превышает 64 Кбайта.

В качестве примера использования указателей far приведем две функции библиотеки среды Си: peek_byte и poke_byte. Функция peek_byte читает байт из любого места памяти, а функция poke_byte записывает байт в любое место памяти. Для обеих функций адрес памяти задается в виде сегмента и смещения. Показано также, как из пары сегмент:смещение получать указатель far.

Пример.

```
/* Функция peek_byte возвращает байт памяти, адрес которого
 * задается в виде адреса сегмента и смещения в сегменте
 */
char peek_byte(segment,offset) /* Определение в старом стиле */
unsigned segment;             /* Адрес сегмента */
unsigned offset;              /* Смещение в сегменте */
{
    char far * far_ptr;       /* Указатель far на тип char */
/* Получить из значений сегмента и смещения указатель far */
    * ((unsigned *) &far_ptr + 1) = segment;
    * ((unsigned *) &far_ptr) = offset;
/* Вернуть байт, адрес которого задается указателем far */
    return *far_ptr;
}

/* Функция poke_byte записывает байт в память по адресу,
 * который задается в виде адреса сегмента и смещения
 * в сегменте */
char poke_byte(segment,offset,byte) /* Определение в */
/* старом стиле */
unsigned segment;                  /* Адрес сегмента */
unsigned offset;                   /* Смещение в сегменте */
char byte;                         /* Байт для записи */
{
    char far * far_ptr;           /* Указатель far на тип char */
/* Получить из значений сегмента и смещения указатель far */
    * ((unsigned *) &far_ptr + 1) = segment;
    * ((unsigned *) &far_ptr) = offset;
/* Записать байт по адресу, который задается указателем
 * far, вернуть его */
    return *far_ptr = byte;
}
```

Адрес huge

Адрес huge так же, как и адрес far, состоит из двух частей: адреса сегмента и смещения (т.е. имеет длину 32 бита). Адрес huge в Си-программах задается указателем huge, объявление которого выглядит как:

```
тип huge * huge_pointer;
```

Хотя указатели far и huge занимают 32 бита и оба позволяют осу-

существовать доступ ко всей памяти, но между ними имеются два существенных различия.

Первое различие заключается в том, что указатель `huge` нормализован, т.е. он содержит максимально допустимое значение адреса сегмента для определяемого им адреса. Так как сегмент всегда начинается на границе, кратной 16 байтам, то значение смещения для нормализованного указателя будет в пределах от 0 до F. Например, нормализованной формой указателя `35d2:1253` (определяющего адрес `36f73`) будет `36f7:0003`. Операции сравнения с указателями `huge` оперируют всеми 32 битами указателя и поэтому дают правильный результат.

Второе различие между указателями `far` и `huge` заключается в том, что для указателей `huge` нет ограничений на изменение значения указателя. Если при изменении указателя `huge` происходит переход через границу 16 байт, то изменяется адрес сегмента. Например, увеличение на 1 указателя `25b0:000f` даст `25b1:0000` и, наоборот, уменьшение на 1 указателя `25b1:0000` даст `25b0:000f`. Эта особенность указателя `huge` позволяет адресовать данные, размер которых превышает 64 Кбайта (занимают более одного сегмента). В языке Си указатели `huge` используются для адресации массивов, размер которых превышает 64 Кбайта. Большие массивы рассматриваются дальше в этой главе.

9.5. СТАНДАРТНЫЕ МОДЕЛИ ПАМЯТИ

Система программирования Си предоставляет пять стандартных моделей памяти: малую, среднюю, компактную, большую и сверх-большую. Метод стандартных моделей памяти является наиболее простым способом управления доступом к коду и данным в основной памяти. В этом случае управление памятью осуществляется через режимы компилятора.

Малая модель памяти

Малая модель памяти используется для большинства обычных Си-программ. Программа с малой моделью памяти занимает только два сегмента по умолчанию: до 64 Кбайта для кода программы и до 64 Кбайта для данных программы. Это значит, что общий размер памяти программы с данной моделью памяти не может превышать 128 Кбайта. Для адресации кода и данных используются только адреса `near`.

Схема организации памяти в программе с малой моделью показана на рис. 12.



Рис. 12. Схема организации памяти в программе малой модели

Средняя модель памяти

Средняя модель памяти используется для Си-программ с большим объемом кода и небольшим объемом данных, т.е. в том случае, если код программы превышает 64 Кбайта (не вмещается в сегмент кода по умолчанию). В этом случае программа должна быть разбита на несколько исходных модулей. Средняя модель памяти обеспечивает один сегмент для данных программы и отдельный сегмент для каждого исходного модуля программы. Это значит, что программа может занимать до 1 Мбайта кода и до 64 Кбайта данных. Поэтому в программах со средней моделью памяти для адресации кода используются адреса far, а для адресации данных — адреса near.

Схема организации памяти в программах со средней моделью памяти показана на рис. 13.

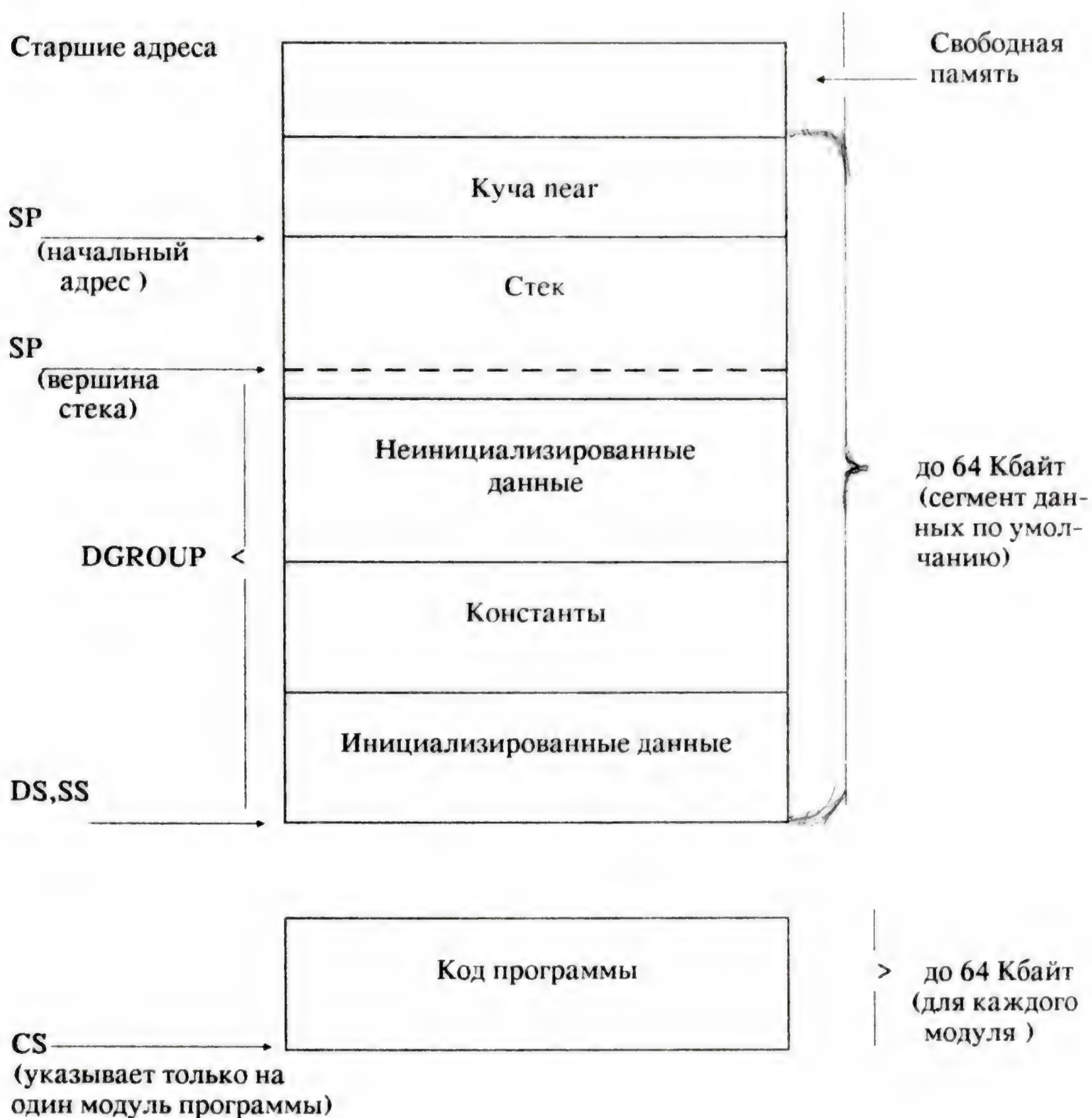


Рис. 13. Схема организации памяти в программе средней модели

Компактная модель памяти

Компактная модель памяти используется для Си-программ с большим объемом данных и небольшим объемом кода. Компактная модель памяти обеспечивает один сегмент для кода программы и несколько сегментов для данных программы. Для инициализированных и неинициализированных глобальных и статических данных программы выделяются отдельные сегменты. Это обеспечивает до 1 Мбайта памяти для глобальных и статических данных. Для адресации кода используются адреса near, а для адресации данных — адреса far.

Схема организации памяти в программах с компактной моделью памяти показана на рис. 14.



Рис. 14. Схема организации памяти в программе компактной модели

Большая модель памяти

Большая модель памяти используется для Си-программ с большим объемом кода и данных. Эта модель памяти обеспечивает несколько сегментов для кода и данных программы. Это гарантирует до 1 Мбайта памяти и для кода, и для данных программы (однако отдельный элемент данных не может превышать 64 Кбайта). В программе с большой моделью памяти используются только адреса far.

Схема организации памяти в программах с большой моделью памяти показана на рис. 15.

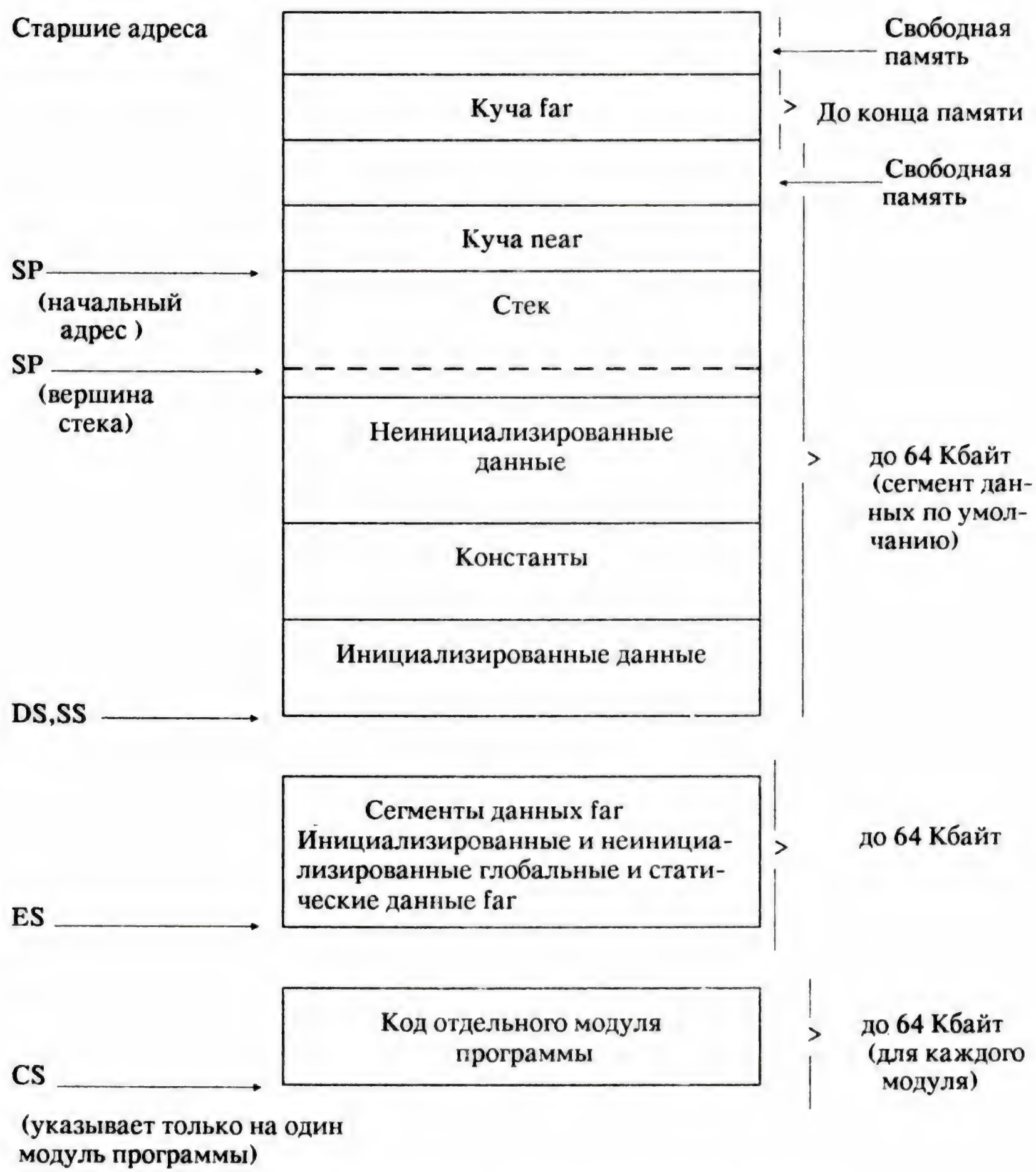


Рис. 15. Схема организации памяти в программе большой или сверхбольшой модели

Сверхбольшая модель памяти

Сверхбольшая модель памяти аналогична большой модели памяти за исключением того, что в сверхбольшой модели памяти снято ограничение на размер отдельного элемента данных. В программе со

сверхбольшой моделью памяти можно использовать массивы, размер которых превышает 64 Кбайта. Для адресации таких массивов используются адреса huge.

Схема организации памяти в программах со сверхбольшой моделью памяти показана на рис. 15.

9.6. РАЗМЕР УКАЗАТЕЛЕЙ
В СТАНДАРТНЫХ МОДЕЛЯХ ПАМЯТИ

Размер указателя на элемент данных или функцию зависит от определенной для программы модели памяти, если только указатель явно не объявлен с одним из ключевых слов near, far или huge. Такое неявное определение размера указателя, характерное для каждой модели памяти, называется определением размера указателя по умолчанию. Размеры указателей на данные и функции, предполагаемые по умолчанию для конкретной модели памяти, приведены в табл. 9.1.

Таблица 9.1

Размер указателей по умолчанию

Модель памяти	Размер указателя на данные	Размер указателя на функцию
Малая	16 бит (адрес near); в адресной арифметике используются 16 бит	16 бит (адрес near)
Средняя	16 бит (адрес near); в адресной арифметике используются 16 бит	32 бита (адрес far)
Компактная	32 бита (адрес far); в адресной арифметике используются 16 бит смещения	16 бит (адрес near)
Большая	32 бита (адрес far); в адресной арифметике используются 16 бит смещения	32 бита (адрес far)
Сверхбольшая	32 бита (адрес huge); в адресной арифметике используются 32 бита	32 бита (адрес far)

Примеры неявного определения размера указателя в зависимости от заданной модели памяти:

```
char *pc, (*pfun)(char);          /* Малая модель */
```

Объявлены два указателя near. Указатель pc является указателем near на величину типа char, а указатель pfun – указателем near на функцию (near), которая получает и возвращает значение типа char.

```
double *pd;          /* Средняя модель */  
double (* pfun)(double);
```

В этом примере pd является указателем near на величину типа

`double`, а `pfun` — указателем `far` на функцию (`far`), получающую и возвращающую значение типа `double`.

```
int * pfi;           /* Компактная модель */
void (* pfun)(int,int);
```

Указатель `pfi` объявлен как указатель `far` на массив целых значений (в отдельные сегменты данных помещаются большие элементы данных, например, массивы). Указатель `pfun` является указателем `near` на функцию (`near`), которая получает два целых значения и не возвращает значений.

```
double *pfd;         /* Большая модель */
void (*pfun)(void);
```

Объявлены два указателя `far`. Один является указателем на массив значений с удвоенной точностью вне сегмента данных по умолчанию, а второй — указателем на функцию (`far`), которая не получает и не возвращает значений.

```
int *phi;            /* Сверхбольшая модель */
int (*pfun)();
```

Указатель `phi` объявлен как указатель `huge` на величину типа `int`. Он позволяет адресовать массив, размер которого превышает 64 Кбайта. Указатель `pfun` является указателем `far` на функцию (`far`), возвращающую значение типа `int`.

9.7. ИЗМЕНЕНИЕ РАЗМЕРА УКАЗАТЕЛЕЙ ДЛЯ СТАНДАРТНЫХ МОДЕЛЕЙ ПАМЯТИ

Одним из недостатков концепции стандартных моделей памяти является то, что при изменении модели памяти изменяются размеры всех адресов данных и кода. Однако можно подавить задаваемый по умолчанию способ адресации для конкретной модели, используя ключевые слова `near`, `far` и `huge`. Эти модификаторы типа могут использоваться со стандартной моделью памяти для того, чтобы избежать ограничений адресации для конкретных элементов данных или кода или для оптимизации доступа к элементам без изменения способа адресации, характерного для программы в целом.

Объявление данных с ключевыми словами `near`, `far` и `huge`

Ключевые слова `near`, `far` и `huge` модифицируют либо размещение объектов, либо размер указателей на объекты. Правила интерпретации этих ключевых слов при использовании их в объявлении данных:

- 1) ключевое слово модифицирует элемент, расположенный непосредственно справа от него;
- 2) если элемент, расположенный непосредственно справа от ключ-

чевого слова, является идентификатором, то ключевое слово определяет, где будет размещен данный элемент: в сегменте данных по умолчанию (near) или в отдельном сегменте данных (far или huge);

3) если элемент, расположенный непосредственно справа от ключевого слова, является указателем на данные, то ключевое слово определяет, будет ли указатель содержать адрес near (16 бит), адрес far (32 бита) или адрес huge (32 бита).

Влияние этих ключевых слов на размещение данных и размер (вид) указателей в зависимости от конкретной модели памяти отражено в табл. 9.2.

Таблица 9.2

Влияние модификаторов на размещение объекта и вид указателей

Модификатор	Размещение объекта данных	Вид указателя
Малая и средняя модели памяти		
Не указан (режим по умолчанию)	В сегменте данных по умолчанию; адресуется 16-битовым адресом	Адрес near
near	Аналогично режиму по умолчанию	Аналогично режиму по умолчанию
far	Вне сегмента данных по умолчанию; адресуется 32-битовым адресом	Адрес far
huge	Вне сегмента данных по умолчанию; может превышать 64 Кбайта; адресуется 32-битовым адресом	Адрес huge
Компактная и большая модели памяти		
Не указан (режим по умолчанию)	Вне сегмента данных по умолчанию; адресуется 32-битовым адресом	Адрес far
near	В сегменте данных по умолчанию; адресуется 16-битовым адресом	Адрес near
far	Аналогично режиму по умолчанию	Аналогично режиму по умолчанию
huge	Вне сегмента данных по умолчанию; может превышать 64 Кбайта; адресуется 32-битовым адресом	Адрес huge
Сверхбольшая модель памяти		
Не указан (режим по умолчанию)	Вне сегмента данных по умолчанию; может превышать 64 Кбайта; адресуется 32-битовым адресом	Адрес huge
near	В сегменте данных по умолчанию; адресуется 16-битовым адресом	Адрес near
far	Вне сегмента данных по умолчанию; не может превышать 64 Кбайта; адресуется 32-битовым адресом	Адрес far
huge	Аналогично режиму по умолчанию	Аналогично режиму по умолчанию

Примеры объявлений данных с ключевыми словами `near`, `far` и `huge` (предполагается, что объявления сделаны в программе малой модели):

<code>int a[1000];</code>	<code>/* Массивы "a" и "b" распределяются */</code>
<code>int near b[1000];</code>	<code>/* в сегменте данных по умолчанию */</code>
<code>char far c[20000];</code>	<code>/* Массив "c" распределяется в */</code>
	<code>/* сегменте данных far */</code>
<code>char huge d[70000];</code>	<code>/* Массив "d" должен быть объявлен как</code>
	<code>huge, так как его размер больше 64 Кбайт.</code>
	<code>Использование ключевого слова huge</code>
	<code>вместо стандартной сверхбольшой модели</code>
	<code>памяти означает, что адреса huge будут</code>
	<code>использоваться для доступа только к мас-</code>
	<code>сиву "d". К остальным данным осуществ-</code>
	<code>ляется доступ внутри сегмента по умолча-</code>
	<code>нию */</code>
<code>int *pa;</code>	<code>/* Указатели pa и pb объявлены */</code>
<code>int near *pb;</code>	<code>/* как указатели near на элемент */</code>
	<code>/* типа char. Указатель pa является */</code>
	<code>/* near по умолчанию */</code>
<code>char far *pc;</code>	<code>/* Указатель pc объявлен как указатель</code>
	<code>far на элемент типа char; может быть ис-</code>
	<code>пользован как указатель на массив симво-</code>
	<code>лов вне сегмента данных по умолчанию,</code>
	<code>например, на массив "c" */</code>
<code>char huge *pd;</code>	<code>/* Указатель pd объявлен как указатель</code>
	<code>huge на элемент типа char; может быть</code>
	<code>использован для ссылки на массив "d" */</code>

Объявление функций с ключевыми словами `near` и `far`

Интерпретация ключевых слов `near` и `far` при использовании их в объявлении функций или указателей на функции (ключевое слово `huge` нельзя применять к функциям) производится по следующим правилам.

1. Ключевое слово модифицирует функцию или указатель, расположенный непосредственно справа от него.

2. Если элемент, расположенный непосредственно справа от ключевого слова, является именем функции, то ключевое слово определяет, будет ли функция распределяться как `near` или `far`.

3. Если элемент, расположенный непосредственно справа от ключевого слова, является указателем на функцию, то ключевое слово определяет, будет ли для вызова функции использоваться адрес `near` (16-битовый) или адрес `far` (32-битовый).

4. Объявление функции должно соответствовать определению функции.

Примеры объявления функций и указателей на функции с ключевыми словами `near` и `far`:

```
int far fun(void);
```

Функция `fun` объявлена как функция, для вызова которой используется 32-битовый адрес и которая возвращает значение типа `int`.

```
int (far * fpf) ();
```

Указатель `fpf` объявлен как указатель `far` на функцию, которая возвращает значение типа `int`.

```
char far * near fun (char far *);
```

Функция `fun` объявлена как функция `near`, которая получает и возвращает указатель `far` на значение типа `char`.

```
int (far * farr[10]) (int far *);
```

Объявлен массив с именем `farr`, состоящий из десяти указателей `far` на функции, которые получают указатель `far` на значение типа `int` и возвращают значение типа `int`.

```
int (far * fun(int (far *) (int far *))) (int far *);
```

Функция `fun` объявлена как функция, которая получает и возвращает указатель `far` на функцию, которая в свою очередь получает указатель `far` на значение типа `int` и возвращает значение типа `int`.

Влияние ключевых слов `near` и `far` на размещение функций и размер (вид) указателей на функции представлено в табл. 9.3.

Пример использования функции `near` в программе с большой моделью памяти:

```
/* Большая модель памяти */
char far * near fun();          /* Объявление функции */
...
char far * near fun()          /* Определение функции */
{
...
}
```

В программе с большой моделью памяти функция `fun` объявлена как функция `near`, возвращающая указатель `far` на значение типа `char`. Такая функция в программе с большой моделью памяти может рассматриваться как функция, которая часто используется, но только функциями текущего модуля. Так как всем функциям данного модуля (содержащего функцию `fun`) распределяется один и тот же сегмент кода, то функция `fun` всегда будет доступна через вызов `near`. Но нельзя передать указатель на функцию `fun` как аргумент другой функции вне модуля, в котором функция `fun` была объявлена.

Таблица 9.3

Влияние модификаторов на размещение функции и вид указателя

Модификатор	Размещение функции	Вид указателя на функцию
Малая и компактная модели памяти		
Не указан (режим по умолчанию) near	В сегменте кода по умолчанию; адресуется 16-битовым адресом Аналогично режиму по умолчанию	Адрес near Аналогично режиму по умолчанию
far	В сегменте кода по умолчанию; адресуется 32-битовым адресом	Адрес far
Средняя, большая и сверхбольшая модели памяти		
Не указан (режим по умолчанию)	Вне сегмента кода по умолчанию; адресуется 32-битовым адресом (неявное объявление функции как far)	Адрес far
near	Вне сегмента кода по умолчанию; адресуется 16-битовым адресом; доступна только функциям текущего модуля	Адрес near
far	Аналогично режиму по умолчанию (явное объявление функции как far)	Аналогично режиму по умолчанию

9.8. ОСОБЕННОСТИ РАБОТЫ С БОЛЬШИМИ МАССИВАМИ

Реализация позволяет использовать массивы, размер которых превышает 64 Кбайта. Для определения таких массивов предназначено ключевое слово `huge`. Например

```
char huge huge_array[70000];      /* Объявление массива huge
                                   размером 70000 байт */
```

Данное объявление массива `huge` используется в программах с малой, средней, компактной и большой моделями памяти. Это явное объявление массива `huge`. В программе со сверхбольшой моделью памяти в объявлении массива размером более 64 Кбайт можно опускать ключевое слово `huge`, массив будет массивом `huge` по умолчанию. Для адресации таких массивов в программе со сверхбольшой моделью памяти используется адрес `huge`.

Элементы массива `huge` размером свыше 64 Кбайт должны удовлетворять следующим условиям:

- 1) элемент массива не может превышать 64 Кбайта;
- 2) если размер массива превышает 128 Кбайт, то элементы такого массива должны иметь размер в байтах, равный степени двух, т.е. 2, 4, 8, 16 байт и т.д.

Применение операции `sizeof` к массивам `huge` и вычитание указателей на массивы `huge` также имеет некоторые особенности.

Использование операции sizeof с массивами huge

Значение, возвращаемое операцией sizeof, определяется в языке как значение типа unsigned int. Размер же массива huge в байтах является величиной типа unsigned long. Это означает, что применение операции sizeof к массиву huge размером более 64 Кбайта будет давать неправильный результат.

Например:

```
char huge huge_arr[70000];  
...  
printf ("Размер массива huge_arr: %ld\n",  
        sizeof(huge_arr));
```

Максимальное значение, которое может быть представлено в типе unsigned int, равно 65535. Размер же массива huge_arr равен 70000 байт, поэтому результатом выражения

sizeof (huge_arr)

будет значение, представляемое младшими 16 битами числа 70000.

Оператор printf напечатает:

Размер массива huge_arr: 4464

Выходом из данной ситуации является использование операции приведения типа:

(unsigned long) sizeof(huge_arr)

Следующий оператор printf даст правильный результат:

```
printf ("Размер массива huge_arr: %ld\n",  
        (unsigned long) sizeof(huge_arr));
```

Вычитание указателей huge

Результат вычитания двух указателей является величиной типа int. Однако при вычитании двух указателей на массив huge (один и тот же) результат может быть величиной типа long int.

Поэтому, как и в случае операции sizeof, чтобы получить правильный результат, необходимо результат вычитания привести к типу long:

```
printf ("Разность двух указателей huge: %ld\n",  
        (long) (huge_ptr2 - huge_ptr1));
```

9.9. ПРЕОБРАЗОВАНИЕ УКАЗАТЕЛЕЙ ПРИ ВЫЗОВЕ ФУНКЦИЙ

Изменение модели памяти может также повлиять на размер указателей, когда они передаются в качестве аргументов функции. Передача указателей в функцию приводит к автоматическому преобразованию размера указателя к наибольшему из следующих двух размеров: размера по умолчанию указателя используемой модели памяти;

размера типа аргумента.

Эти преобразования выполняются в следующих случаях:

если программист не задал прототип вызываемой функции с типами передаваемых ей указателей;

если программист задал прототип вызываемой функции, но без списка типов передаваемых функции указателей.

Если же задан прототип функции с типами передаваемых ей указателей, то при вызове функции будут выполняться преобразования указателей к типам, указанным в прототипе функции. Отсутствие прототипов функций может приводить к неправильному выполнению программы. Например, следующая программа будет правильно работать с малой и средней моделями памяти, но будет давать неожиданные результаты с компактной, большой и сверхбольшой моделями памяти:

```
#include <stdio.h>
main()
{
    char *p1, *p2;
    int rc;
    p1 = "Это строка 1";
    p2 = "Это строка 2";
    rc = compare(p1, p2);
    if (rc < 0)
        printf("Строка 1 лексикографически меньше строки 2\n");
    else if (rc > 0)
        printf("Строка 1 лексикографически больше строки 2\n");
    else
        printf("Строки лексикографически равны\n");
}

int compare(char near *s1, char near *s2)
{
    for (; *s1 == *s2; s1++, s2++)
        if (!*s1)
            return (0);
    return (*s1 - *s2);
}
```

В этой программе отсутствует прототип функции `compare`. Поэтому, если программа используется с малой или средней моделью памяти, указатели `p1` и `p2` будут представлены как указатели `near` (16 бит) перед передачей в функцию `compare`. Так как параметры `s1` и `s2` в функции `compare` объявлены как указатели `near`, то будет выполняться правильное присваивание аргументов `p1` и `p2` параметрам `s1` и `s2`. Следовательно, программа будет работать правильно. Однако, если программа используется с компактной, большой или сверхбольшой моделью памяти, указатели `p1` и `p2` будут представлены как указатели `far` (32 бита). Параметры `s1` и `s2` определены как указатели `near`. Поэтому при вызове функции `compare` параметр `s1` получит первые 16 бит

передаваемого 32-битового указателя `p1`, параметр `s2` получит оставшиеся 16 бит передаваемого указателя `p1`. Ясно, что это приведет к неправильному выполнению программы. Описанный процесс не приведет и к ошибкам компиляции, так как и вызов, и определение функции `compare` являются правильными конструкциями. Для того чтобы передать `p1` и `p2` как указатели `near`, необходимо включить в программу прототип функции `compare`, в котором аргументы, передаваемые функции, явно объявлены как указатели `near`. Это сделано в следующем варианте программы:

```
#include <stdio.h>
main()
{
    int compare(char near *,char near * );
    char *p1, *p2;
    int rc;
    p1 = "Это строка 1 ";
    p2 = "Это строка 2";
    rc = compare(p1, p2);
    if (rc < 0)
        printf("Строка 1 лексикографически меньше строки 2\n");
    else if (rc > 0)
        printf("Строка 1 лексикографически больше строки 2\n");
    else
        printf("Строки лексикографически равны\n");
}
int compare(char near *s1, char near *s2)
{
    for (; *s1 == *s2; s1++, s2++)
        if (!*s1)
            return (0);
    return (*s1 - *s2);
}
```

Глава 10

ОРГАНИЗАЦИЯ СВЯЗИ С ПРОГРАММАМИ НА ДРУГИХ ЯЗЫКАХ

Система программирования Си позволяет осуществлять связь с программами на языках ассемблер, Паскаль, Фортран и Бейсик. Здесь будут рассмотрены правила написания программ на языках ассемблер, Паскаль и Фортран для использования с программами на языке Си, а также элементы языка Си для организации интерфейса с этими языками.

Реализация позволяет встраивать в текст Си-программы команды ассемблера. Для этой цели предназначено специальное ключевое сло-

во `_asm`, которое отмечает начало встраиваемого ассемблерного кода. Ассемблерный код записывается в следующем формате:

```
_asm <команда языка ассемблера> ;
```

если встраивается одна ассемблерная команда и

```
_asm {  
    команды языка ассемблера  
}
```

если встраивается блок ассемблерных команд.

Пример использования встроенного в Си-программу ассемблерного кода можно найти в п. 11.6.

Программой на другом языке здесь будет называться любая функция, подпрограмма или процедура на ассемблере, Паскале или Фортране, которая может быть вызвана из программы на языке Си (табл. 10.1).

Таблица 10.1

Программные единицы межязыковых связей

Язык	Программная единица	
	есть возвращаемое значение	нет возвращаемого значения
Си	Функция	Функция void
Ассемблер	Процедура	Процедура
Паскаль	Функция	Процедура
Фортран	Функция	Подпрограмма

Для вызова из Си-программы программы на другом языке необходимо, чтобы и вызывающая программа на Си, и вызываемая программа на другом языке были написаны с использованием совместимых соглашений о связях и именах. Языки Паскаль и Фортран используют одинаковые соглашения о связях и именах, которые по умолчанию отличаются от соглашений о связях и именах Си. Однако эти языки позволяют использовать две формы соглашений о связях: соглашения о связях самого языка (соглашения по умолчанию) и соглашения о связях Си (альтернативная форма соглашений). Язык ассемблер также содержит элементы языка, позволяющие использовать различные соглашения о связях.

При организации связи Си-программ с программами на языке ассемблера необходимо обеспечить, чтобы в начале выполнения комплекса программ был выполнен код запуска (инициализации), который обеспечивается компилятором Си перед функцией `main`. Если головным модулем комплекса программ является Си-программа с функцией `main`, такая инициализация будет обеспечена автоматически.

Рассматриваемые здесь средства языков ассемблер, Паскаль и Фор-

тран для интерфейса с Си реализованы в соответствующих компиляторах операционной системы Альфа-ДОС для ПЭВМ. В других компиляторах они могут быть не реализованы.

10.1. СОГЛАШЕНИЯ О СВЯЗЯХ И ИМЕНАХ В СИ

1. Аргументы, передаваемые вызываемой программе, помещаются в стек в порядке, обратном тому, в котором они заданы в операторе вызова.

Это означает, что соглашения о связях Си допускают вызов программы с переменным числом аргументов. Так как первый аргумент помещается в стек последним, его адрес в вызываемой программе всегда известен.

2. Аргументы в вызываемую программу передаются по значению, за исключением массивов, которые передаются ссылкой.

3. После завершения вызываемой программы стек восстанавливает вызывающая программа.

4. Компилятор Си в начало любого внешнего имени добавляет символ подчеркивания перед помещением его в объектный файл.

5. Компилятор Си не преобразует строчные буквы внешнего имени в прописные перед помещением его в объектный файл.

6. Компилятор Си распознает первые 31 символ внешнего имени. Если во внешнем имени содержится больше символов, то лишние символы отбрасываются перед помещением имени в объектный файл.

10.2. СОГЛАШЕНИЯ О СВЯЗЯХ И ИМЕНАХ В ПАСКАЛЕ И ФОРТРАНЕ

1. Аргументы, передаваемые вызываемой программе, помещаются в стек в порядке их задания в точке вызова.

Это означает, что соглашения о связях в Паскале и Фортране не позволяют вызывать программы с переменным числом аргументов.

2. Аргументы в вызываемую программу передаются по значению, однако можно определить передачу параметров по ссылке.

Фортран определяет иной способ передачи параметров: параметры в вызываемую программу передаются по ссылке, однако можно определить передачу параметров по значению.

3. При завершении вызываемой программы стек восстанавливает сама вызываемая программа.

4. Строчные буквы внешнего имени преобразуются в прописные перед помещением имени в объектный файл.

5. Компиляторы языков Паскаль и Фортран распознают разное количество символов внешнего имени. Компилятор Паскаля распознает первые восемь символов внешнего имени, а компилятор Фортрана —

первые шесть. Лишние символы отбрасываются перед помещением имени в объектный файл.

6. В отличие от компилятора языка Си компиляторы Паскаля и Фортрана не помещают символ подчеркивания в начало внешнего имени.

10.3. ОРГАНИЗАЦИЯ СВЯЗИ С ПРОГРАММАМИ НА ЯЗЫКЕ АССЕМБЛЕРА

В Си-программе ассемблерная процедура вызывается так же, как внешняя функция Си. При этом могут быть использованы соглашения о связях и именах как предполагаемые по умолчанию в Си (вызов в стиле Си), так и не в стиле Си (вызов в стиле Паскаля).

Вызов ассемблерной процедуры в стиле Си

Для вызова ассемблерной процедуры в стиле Си нужно выполнить следующее.

1. Объявить вызываемую ассемблерную процедуру как внешнюю функцию Си, например,

```
extern void asm_proc (int, int, int);
```

если процедура не возвращает значения, или

```
extern int asm_proc (int, int, int);
```

если процедура возвращает значение.

2. Написать ассемблерную процедуру с соблюдением соглашений о связях и именах в Си.

3. Вызвать ассемблерную процедуру как обычную функцию Си, например,

```
extern void asm_proc (int, int, int);
```

```
int a, b, c;
```

```
...
```

```
asm_proc (a, b, c);
```

если ассемблерная процедура не возвращает значения, или

```
extern int asm_proc (int, int, int);
```

```
int a, b, c, r;
```

```
...
```

```
r=asm_proc (a, b, c);
```

если ассемблерная процедура возвращает значение.

Вызов ассемблерной процедуры в стиле Паскаля

Для вызова ассемблерной процедуры в стиле Паскаля нужно выполнить следующее.

1. Объявить вызываемую ассемблерную процедуру как внешнюю функцию с ключевым словом `pascal` (`fortran`), например,


```
extern void pascal asm_proc (int, int);
```

если процедура не возвращает значения, или

```
extern int pascal asm_proc (int, int);
```

если процедура возвращает значение.

2. Написать ассемблерную процедуру с соблюдением соглашений о связях в Паскале.

3. Вызвать ассемблерную процедуру как функцию Си:

```
extern void pascal asm_proc (int, int);
```

```
int a, b;
```

```
...
```

```
asm_proc (a, b);
```

если процедура не возвращает значения, или

```
extern int pascal asm_proc (int, int);
```

```
int a, b, r;
```

```
...
```

```
r=asm_proc (a, b);
```

если процедура возвращает значение.

Если Си-программа компилируется с малой или компактной моделью памяти, то ассемблерная процедура неявно объявляется как `near`, т.е. для вызова процедуры будет использоваться вызов `near`. Если Си-программа компилируется со средней, большой или сверхбольшой моделью памяти, то ассемблерная процедура неявно объявляется как `far`. Явное использование ключевых слов `near` и `far` в объявлении ассемблерной процедуры позволяет задать требуемую форму вызова процедуры.

Вызов ассемблерной процедуры через указатель:

```
extern void asm_proc (int, int);
```

```
void (* pproc) (int, int) = asm_proc;
```

```
int a, b;
```

```
...
```

```
(*pproc) (a,b);
```

10.4. НАПИСАНИЕ АССЕМБЛЕРНОЙ ПРОЦЕДУРЫ

Схема написания ассемблерной процедуры, вызываемой из Си-программы, должна быть следующей.

1. Ассемблерная процедура должна быть оформлена таким образом, чтобы в ней использовались совместимые с Си-программой сегменты и чтобы форма возврата из процедуры не противоречила форме ее вызова в Си-программе.

2. Кроме основной задачи, выполняемой процедурой, она должна включать следующую стандартную последовательность шагов:

вход в процедуру;
 размещение локальных данных, если необходимо;
 сохранение регистров;
 доступ к параметрам;
 возврат значения, если необходимо;
 выход из процедуры.

Все другие языки высокого уровня, поддерживаемые в операционной системе Альфа-ДОС, предъявляют аналогичные требования к ассемблерной процедуре, что и язык Си. Поэтому ассемблерная процедура, написанная в соответствии с предложенным методом, может быть использована не только в Си-программе.

Оформление ассемблерной процедуры

Для того чтобы ассемблерная процедура могла быть объединена с Си-программой на этапе редактирования, необходимо, чтобы в ней использовались сегменты, совместимые с применяемыми в Си-программе. Имена, типы и классы (по умолчанию) сегментов, которые используются в Си-программе, приведены в табл. 10.2.

Использование обобщенных директив ассемблера MASM избавляет пользователя от необходимости знать характеристики сегментов. Обобщенные ассемблерные директивы, которые могут применяться при организации связи с Си-программой, приведены в табл. 10.3.

Таблица 10.2

Характеристики сегментов Си-программы

Сегмент (содержимое сегмента)	Имя	Тип выравнивания	Тип объединения	Класс
Код модуля	_TEXT или имя_TEXT	WORD	PUBLIC	'CODE'
Постоянные данные	CONST	WORD	PUBLIC	'CONST'
Инициализированные данные near	_DATA	WORD	PUBLIC	'DATA'
Неинициализированные данные near	_BSS	WORD	PUBLIC	'BSS'
Инициализированные данные far	FAR_DATA	PARA	частный	'FAR_DATA'
Неинициализированные данные far	FAR_BSS	PARA	частный	'FAR_BSS'
Неинициализированные данные huge	HUGE_BSS	PARA	частный	'HUGE_BSS'
Стек	STACK	PARA	STACK	'STACK'

Таблица 10.3

Обобщенные директивы ассемблера, характеризующие сегменты

Директива	Сегмент (содержимое сегмента)
.CODE	Код модуля
.DATA	Инициализированные данные
.DATA?	Неинициализированные данные
.FARDATA и .FARDATA?	Данные, которые не будут объединены с соответствующими сегментами в других модулях
.CONST	Постоянные данные (строковые константы или константы с плавающей точкой)
.STACK	Стек (обычно этот сегмент объявляется в головном модуле и не переобъявляется)

Соответствие между обобщенными директивами и именами по умолчанию для каждой модели памяти приведено в табл. 10.4.

Таблица 10.4

Соответствие между обобщенными директивами и сегментами

Модель памяти	Директива	Имя	Тип выравнивания	Тип объединения	Класс	Группа
Малая	.CODE	_TEXT	WORD	PUBLIC	'CODE'	DGROUP
	.DATA	_DATA	WORD	PUBLIC	'DATA'	
	.CONST	_CONST	WORD	PUBLIC	'CONST'	
	.DATA?	_BSS	WORD	PUBLIC	'BSS'	
	.STACK	_STACK	PARA	STACK	'STACK'	
Средняя	.CODE	имя_TEXT	WORD	PUBLIC	'CODE'	DGROUP
	.DATA	_DATA	WORD	PUBLIC	'DATA'	
	.CONST	_CONST	WORD	PUBLIC	'CONST'	
	.DATA?	_BSS	WORD	PUBLIC	'BSS'	
	.STACK	_STACK	PARA	STACK	'STACK'	
Компактная	.CODE	_TEXT	WORD	частный	'CODE'	DGROUP
	.FARDATA	_FAR_DATA	PARA	частный	'FAR_DATA'	
	.FARDATA?	_FAR_BSS	PARA	PUBLIC	'FAR_BSS'	
	.DATA	_DATA	WORD	PUBLIC	'DATA'	
	.CONST	_CONST	WORD	PUBLIC	'CONST'	
Большая или сверх-большая	.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
	.STACK	_STACK	PARA	STACK	'STACK'	
	.CODE	имя_TEXT	WORD	PUBLIC	'CODE'	
	.FARDATA	_FAR_DATA	PARA	частный	'FAR_DATA'	
	.FARDATA?	_FAR_BSS	PARA	частный	'FAR_BSS'	

Если же для определения сегментов используются директивы **SEGMENT**, **ENDS** и **ASSUME**, то необходимо использовать указанные имена, типы выравнивания, типы объединения и классы. Например, сегмент кода для малой модели памяти должен быть объявлен так:

```
_TEXT SEGMENT WORD PUBLIC 'CODE'
```

Ассемблерный текст следует оформлять в виде процедуры с директивой **PROC**, которая упрощает написание ассемблерной программы. Она автоматически обеспечивает:

- генерацию команд входа в процедуру и выхода из процедуры;
- сохранение регистров на входе в процедуру и восстановление их на выходе;

доступ к принимаемым аргументам посредством текстовых макроопределений (не надо использовать сложные конструкции доступа к параметрам с использованием регистра **BP**).

Директива **PROC** имеет вид:

```
имя-процедуры PROC [NEAR|FAR]  
[USES [список-сохраняемых-регистров],]  
[аргумент:[ [NEAR|FAR] PTR] тип]  
[,...]
```

где *типом* аргумента может быть **WORD**, **DWORD**, **FWORD** или имя структуры, определенной директивой **STRUCT**.

В начале исходного файла можно использовать директиву **.MODEL**. Эта директива позволяет:

- задавать необходимую модель памяти;
- устанавливать необходимые соглашения о связях и именах;
- генерировать подходящие команды возврата.

Можно не объявлять имя процедуры глобальным (директивой **PUBLIC**), если задана директива **.MODEL**, устанавливающая соглашения о связях и именах.

Директива **.MODEL** имеет вид:

```
.MODEL модель-памяти [,язык]
```

Например, директива

```
.MODEL SMALL, C
```

означает, что будут использоваться соглашения о связях и именах **Си**.

Вход в процедуру

На входе в процедуру верхняя часть стека имеет вид, приведенный на рис. 16.

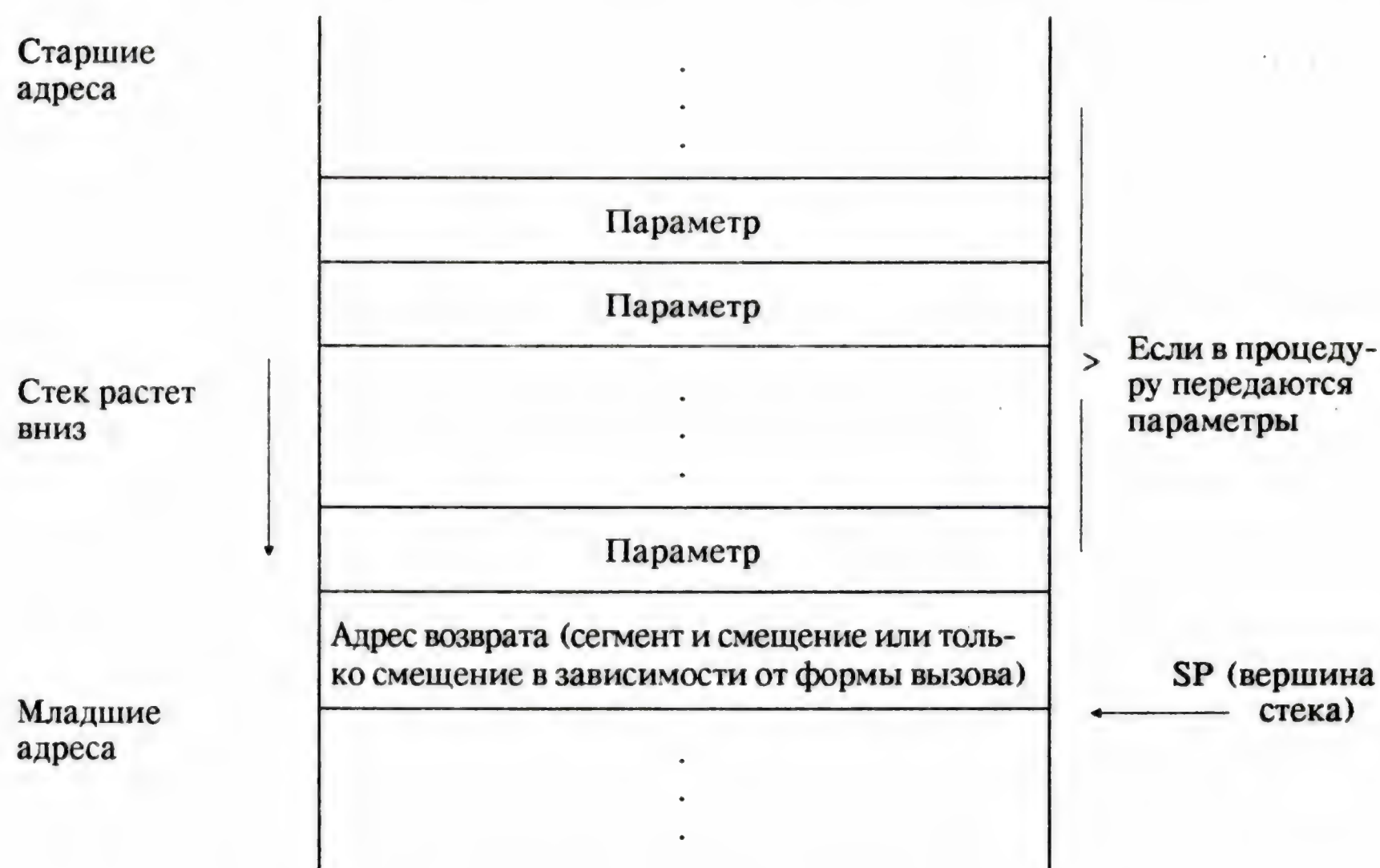


Рис. 16. Верхняя часть стека при входе в процедуру

Как видно из рис. 16, для того чтобы процедура имела доступ к передаваемым параметрам, необходимо зафиксировать значение указателя вершины стека (регистр SP), которое он имел на входе в процедуру. Регистр SP не может быть использован для доступа к параметрам, так как он не является ни индексным, ни базовым регистром. Кроме того, значение регистра SP может изменяться в процессе работы процедуры. Для доступа к параметрам предназначен базовый регистр BP. Поэтому процедуру следует начинать со следующих команд:

```
push bp
mov bp, sp
```

Сначала необходимо сохранить значение регистра BP из вызывающей программы, а затем зафиксировать значение указателя вершины стека на входе в процедуру. Значение регистра BP будет оставаться постоянным на протяжении всей процедуры и поэтому каждый параметр может быть адресован фиксированным смещением от BP (рис. 17).

Если ассемблерный текст оформлен в виде процедуры PROC, то команды входа в процедуру можно опускать.

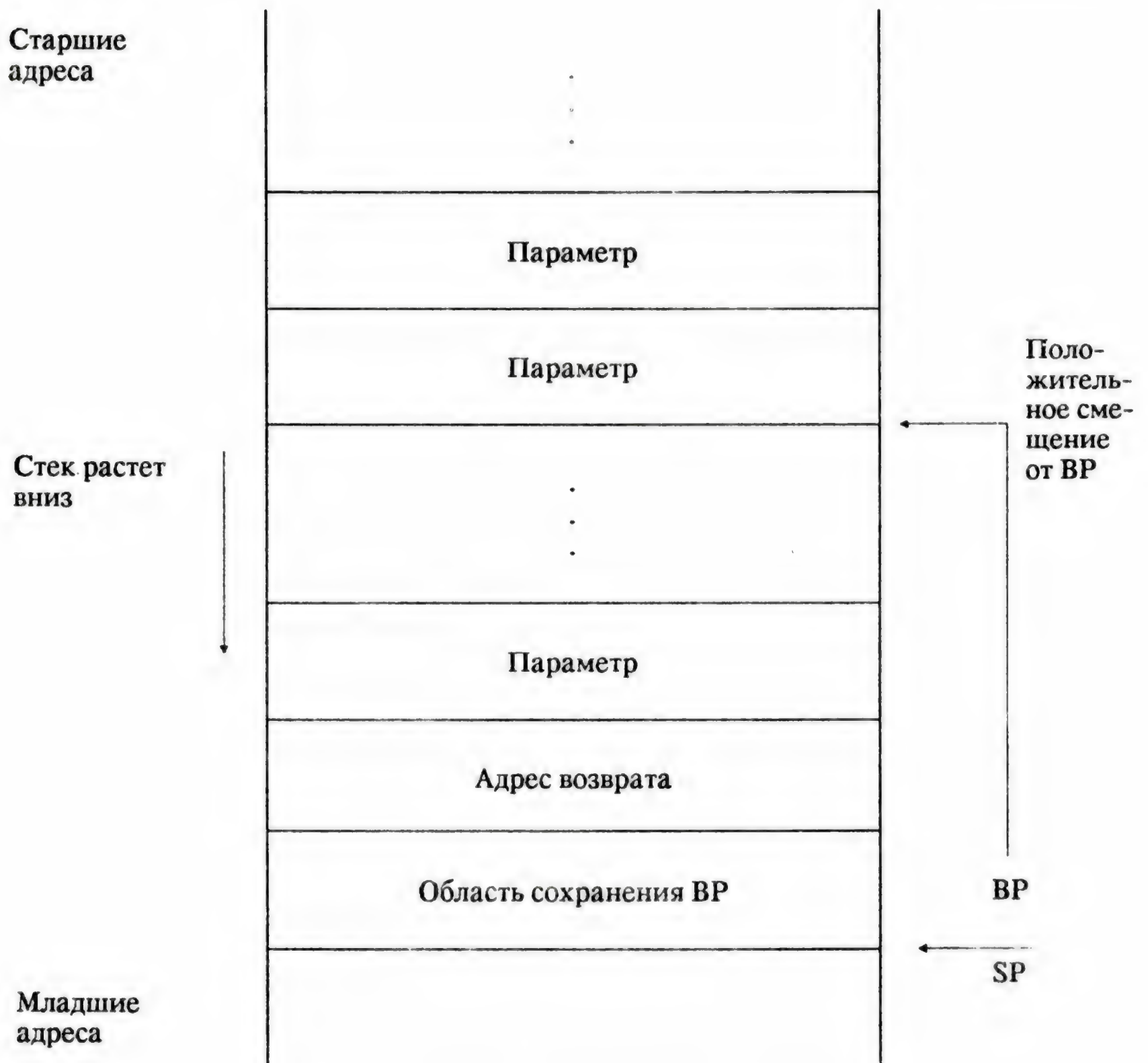


Рис. 17. Адресация параметров в процедуре

Размещение локальных данных

Если в ассемблерной процедуре будут использоваться локальные данные, то следующим шагом при написании процедуры должно быть резервирование области под эти данные. Область под локальные данные отводится в стеке. Для этого необходимо значение регистра SP уменьшить на размер области локальных данных:

```
sub sp, <размер области в байтах>
```

Регистр SP всегда нужно уменьшать на четное количество байтов. После этой команды верхняя часть стека будет иметь вид, представленный на рис. 18.

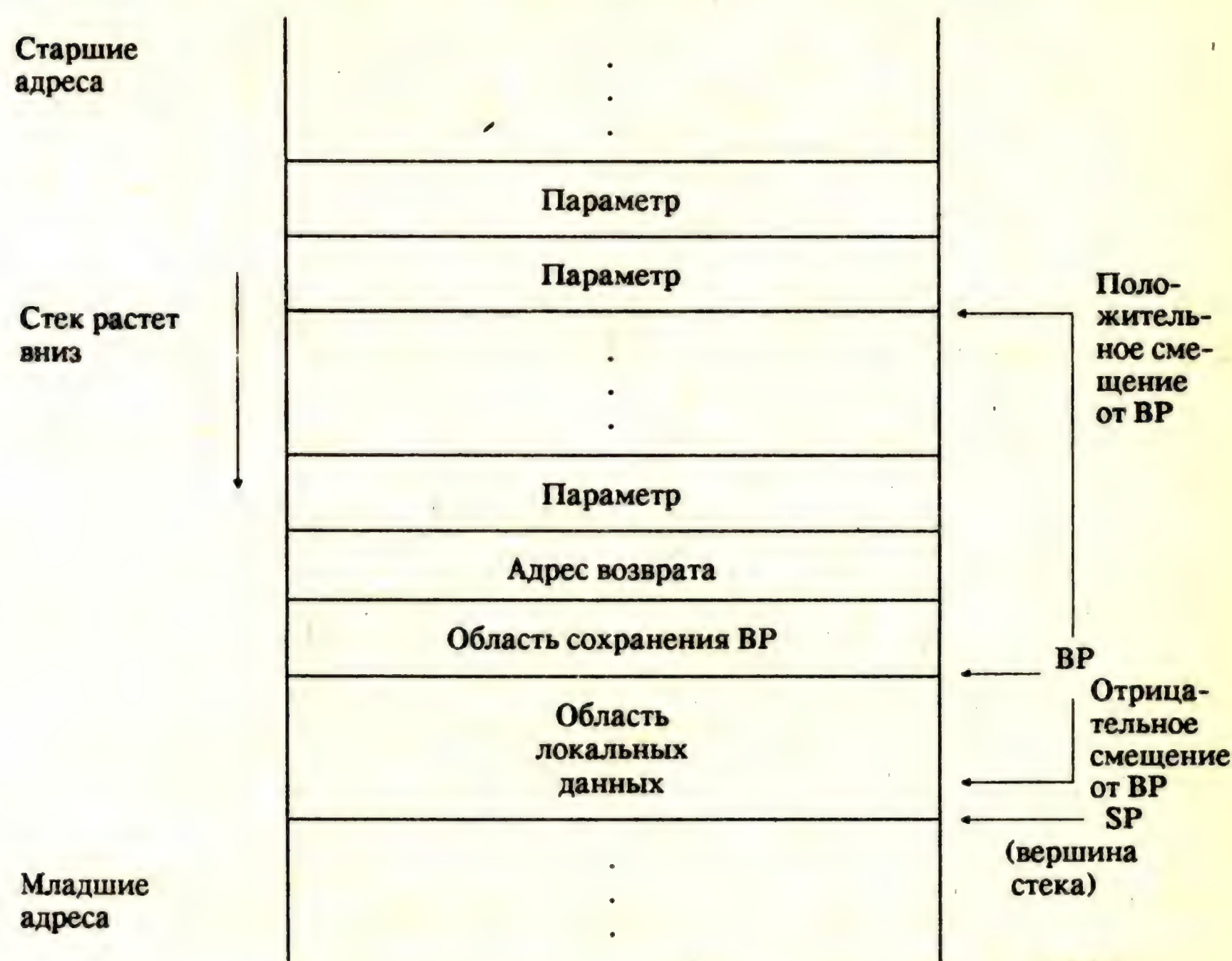


Рис. 18. Размещение локальных данных в стеке

Локальные данные адресуются фиксированным отрицательным смещением от BP, например,

```
mov WORD PTR [bp-2], 0
mov WORD PTR [bp-4], 0
```

Область стека, отведенная под локальные данные процедуры, должна быть восстановлена в конце процедуры.

Работу с локальными данными можно упростить, если использовать директиву ассемблера LOCAL. Директива LOCAL позволяет распределять локальные переменные и определять для доступа к ним текстовые макроопределения.

```
Директива имеет вид:
LOCAL переменная[[число]][:[[NEAR|FAR]PTR]тип]
[,...]
```

```
Например,
LOCAL lvar1:WORD, lvar2:WORD
```

```
...
mov lvar1, 0
mov lvar2, 0
```


Эта последовательность команд эквивалентна следующей последовательности:

```
lvar1 EQU <WORD PTR [bp-2]>
lvar2 EQU <WORD PTR [bp-4]>
...
sub sp, 4
...
mov lvar1, 0
mov lvar2, 0
```

Следует помнить о том, что локальные переменные автоматически не инициализируются.

Сохранение регистров

Следующим шагом при написании ассемблерной процедуры является сохранение регистров, которые будут изменяться в процессе работы ассемблерной процедуры. Обычно ассемблерная процедура, вызываемая из программы на любом языке высокого уровня, должна сохранять (в стеке) значения регистров SI, DI, DS и SS (в дополнение к регистру BP). Если же процедура не меняет значения этих регистров, то регистры можно не сохранять. Перед выходом из процедуры необходимо восстановить значения сохраненных регистров.

Например,

```
...
push si      ; Сохранить SI и DI
push di
...
pop di       ; Восстановить DI и SI
pop si
```

Можно не сохранять регистры, если имена сохраняемых регистров указаны в заголовке процедуры, например,

```
asm_proc PROC NEAR USES SI DI
```

Доступ к параметрам

После того как зафиксирован указатель вершины стека на входе в процедуру, размещены локальные данные и сохранены необходимые регистры, можно приступать к написанию основного тела процедуры. Доступ к параметрам в теле процедуры осуществляется через регистр BP с положительным смещением. По соглашениям о связях в языке Си параметры, передаваемые в процедуру, сохраняются в стеке в порядке, обратном их заданию в точке вызова в исходной программе. Например,

```
extern int asm_proc (int, int, int);
int a, b, c, r
...
r=asm_proc (a,b,c);
```


Перед вызовом процедуры `asm_proc` аргументы "a", "b", "c" запоминаются в стеке в таком порядке: c, b, a. Верхняя часть стека на входе в процедуру `asm_proc` будет иметь вид, представленный на рис. 19.

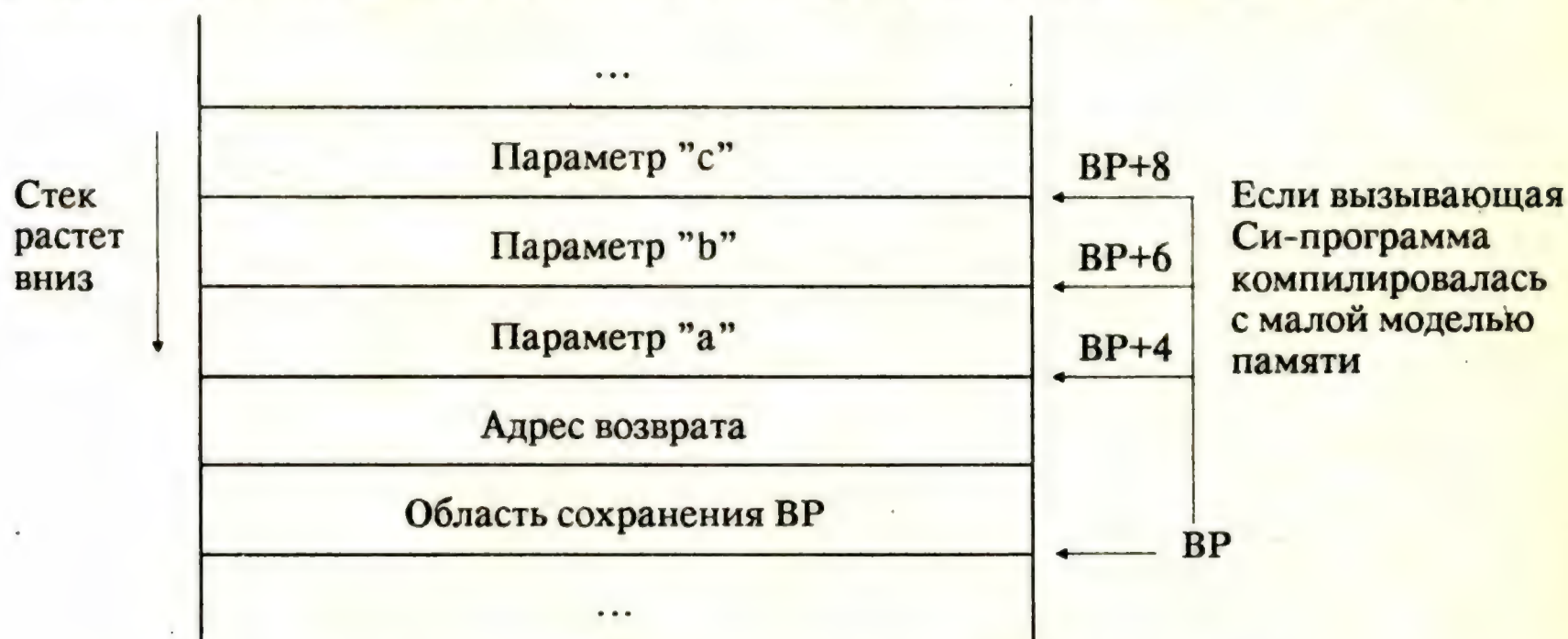


Рис. 19. Размещение аргументов в стеке (вызов в стиле Си)

В теле процедуры `asm_proc` параметр "a" имеет адрес `BP+4`, параметр "b" — `BP+6` и параметр "c" — `BP+8`.

Если вызов ассемблерной процедуры осуществляется в стиле Паскаля, например,

```
extern int pascal asm_proc (a, b, c);
int a, b, c, r;
...
r=asm_proc (a, b, c);
```

аргументы "a", "b" и "c" запоминаются в стеке в том же порядке, как они заданы в точке вызова. Верхняя часть стека при входе в процедуру в этом случае будет выглядеть, как показано на рис. 20.

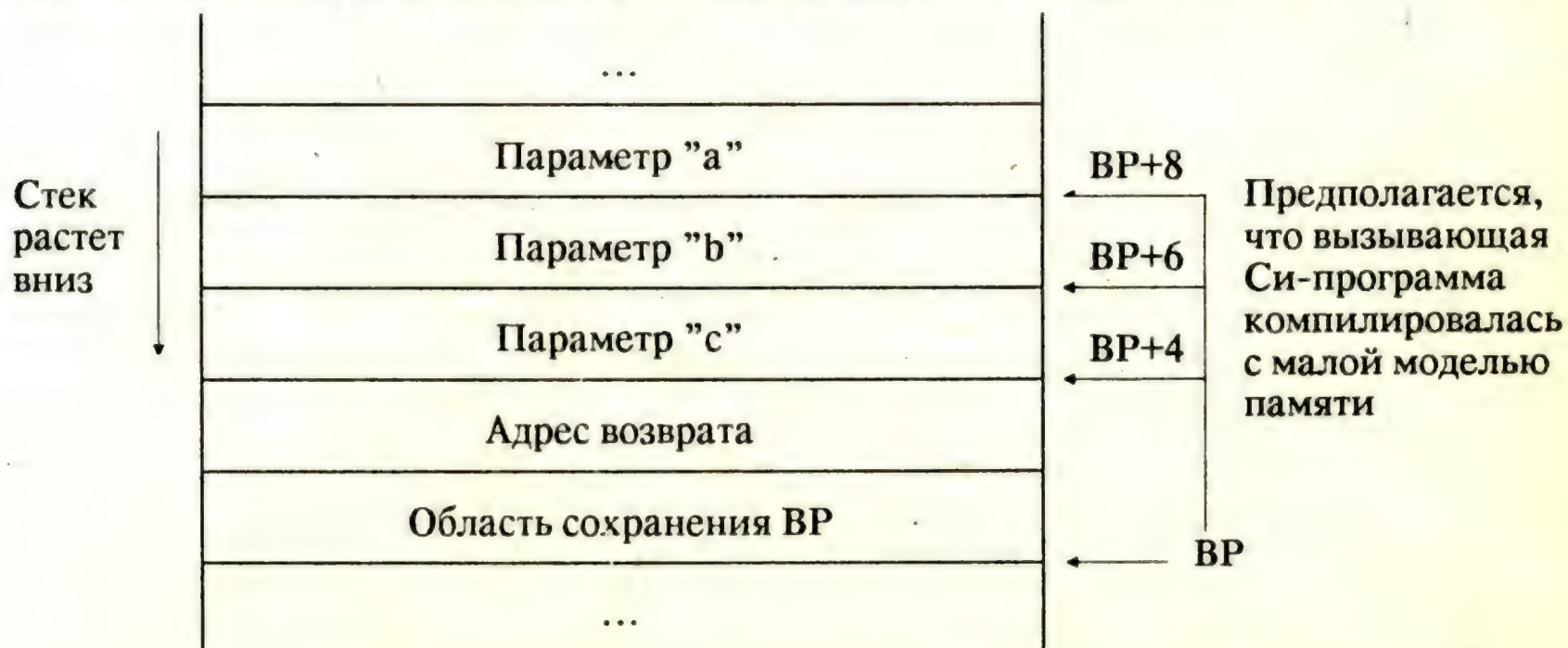


Рис. 20. Размещение аргументов в стеке (вызов в стиле Паскаля)

В теле процедуры `asm_proc` для доступа к параметрам "а", "b" и "с" используются адреса `BP+8`, `BP+6` и `BP+4` соответственно.

Работу с параметрами в теле процедуры можно упростить, если параметры, которые передаются процедуре, указать в заголовке процедуры, например,

```
.MODEL SMALL, C
```

```
asm_proc PROC USES SI DI, a:WORD, b:WORD, c:WORD
```

если параметры передаются в стиле Си, или

```
.MODEL SMALL, PASCAL
```

```
asm_proc PROC USES SI DI, a:WORD, b:WORD, c:WORD
```

если параметры передаются в стиле Паскаля.

Параметры "а", "b" и "с" в этом случае определяются по умолчанию как следующие текстовые макроопределения:

если передача параметров в стиле Си:

```
a EQU <WORD PTR [bp+4]>
```

```
b EQU <WORD PTR [bp+6]>
```

```
c EQU <WORD PTR [bp+8]>
```

если передача параметров в стиле Паскаля:

```
a EQU <WORD PTR [bp+8]>
```

```
b EQU <WORD PTR [bp+6]>
```

```
c EQU <WORD PTR [bp+4]>
```

Использование параметров в теле процедуры теперь намного упрощается, например,

```
mov ax, a
```

```
add ax, b
```

```
add ax, c
```

Возврат значения

Если ассемблерная процедура возвращает значение, то для возврата значения в Си-программу необходимо придерживаться следующих правил.

1. Если возвращаемое значение имеет основной или адресный тип (не массив и не структурный тип) и не длиннее четырех байт, то значение должно возвращаться, как описано ниже.

2. Если возвращаемое значение больше четырех байт, то ассемблерная процедура должна зарезервировать место для возвращаемого значения и вернуть его адрес в паре регистров `DX:AX` (либо только в `AX`).

Правила возврата значений

Размер значения в байтах	Регистр, в котором возвращается значение
1	AL
2	AX (через регистр AX возвращаются и адреса near)
4	DX:AX (через пару регистров DX:AX возвращаются и адреса far) — DX для старшей части значения (или адреса сегмента) — AX для младшей части значения (или адреса смещения)

Выход из процедуры

Перед выходом из ассемблерной процедуры необходимо восстано-
вить все сохраненные регистры.

1. Если сохранялись какие-либо из регистров SS, DS, SI и DI, вос-
становить их в порядке, обратном тому, в котором они сохранялись.
Например, если в начале процедуры сохранялись регистры SI и DI

```
push si
push di
```

то в конце процедуры их необходимо восстановить следующими ко-
мандами:

```
pop di
pop si
```

2. Восстановить регистр SP, если в начале процедуры распределя-
лась область под локальные данные:

```
mov sp, bp
```

3. Восстановить регистр BP:

```
pop bp
```

Если ассемблерная программа оформлена в виде процедуры PROC, то команды восстановления пп. 2 — 3 можно опускать. Если, кроме того, в заголовке процедуры заданы имена сохраняемых регистров, то команды сохранения и восстановления этих регистров генерируются ассемблером автоматически, поэтому их тоже можно опускать.

Последним шагом ассемблерной процедуры должен быть выход в Си-программу по команде `ret` (`retn` или `retf`).

Обычно программист должен восстанавливать стек при выходе из процедуры, если процедура была вызвана с использованием соглаше-
ний о связях, не поддерживаемых Си. Например,

```
ret 6 ;
```

В процедуру передавались три целых параметра.

Если задана директива `.MODEL` с указанием используемых согла-

шений о связях и именах, то программисту не надо заботиться о восстановлении стека. Ассемблер сгенерирует надлежащую команду выхода из процедуры.

Примеры

Рассмотрим два примера программ на Си, в которых для связи с ассемблерной процедурой используются различные соглашения о связях. Каждая программа состоит из двух модулей: головного модуля на Си и ассемблерной процедуры, которые объединяются вместе на этапе редактирования связей.

Пример 1 представляет программу вычисления среднего значения последовательности целых положительных чисел. Для вычисления среднего значения Си-программа вызывает ассемблерную процедуру, передавая ей последовательность целых положительных чисел.

Этой программой демонстрируются:

вызов ассемблерной процедуры с использованием соглашений о связях и именах Си;

передача переменного числа аргументов в ассемблерную процедуру;

разработка процедуры с использованием и без использования обобщенных директив сегментов и средств интерфейса с языками высокого уровня;

возврат значения из ассемблерной процедуры в Си-программу.

Пример 1.

Головной модуль на Си:

```
#include <stdio.h>
main()
{
    /* Объявление ассемблерной процедуры с использованием
       соглашений о связях и именах Си */
    extern int near average(int, ...);
    int n;
    /* Вызов процедуры near в стиле Си */
    n = average(10, 3, 5, 2, 12, -1);
    printf("Среднее = %d\n", n); /* Печатается число 6 */
}
```

Вариант ассемблерной процедуры average, демонстрирующий соблюдение соглашений о связях Си:

```
; Процедура должна быть проассемблирована с одним из
; режимов /Mx или /Ml, чтобы избежать преобразования
; имени процедуры в прописные буквы
;
_TEXT SEGMENT WORD PUBLIC 'CODE'
    ASSUME cs:_TEXT
    PUBLIC _average
_average PROC
    push bp    ; Сохранить BP из Си-программы
    mov bp, sp ; Зафиксировать указатель вершины стека
```



```

    sub sp, 6    ; Отвести место под локальные
                  ; переменные
    push si      ; Сохранить SI
                  ; Очистить локальные переменные
    mov WORD PTR [bp-2], 0 ; Сумма (старшая часть)
    mov WORD PTR [bp-4], 0 ; Сумма (младшая часть)
    mov WORD PTR [bp-6], 0 ; Счетчик числа параметров
    xor si, si    ; SI используется как индекс
                  ; в списке параметров
comp_sum:        ; Начало цикла для подсчета
                  ; суммы всех целочисленных
                  ; параметров
    mov ax, WORD PTR [bp + si + 4] ; Получить очередной
                  ; параметр
    cmp ax, -1    ; Если это признак конца
    je exit       ; списка параметров пере-
                  ; менной длины —
                  ; выйти из цикла
    add WORD PTR [bp-4], ax ; Добавить параметр к сумме
    adc WORD PTR [bp-2], 0 ; Учесть перенос в старшую
                  ; часть
    inc WORD PTR [bp-6]    ; Увеличить счетчик параметров
    add si, 2              ; Увеличить индекс
    jmp comp_sum
exit:
    mov dx, WORD PTR [bp-2] ; Вычислить среднее число и
                  ; вернуть его
    mov ax, WORD PTR [bp-4] ; в регистре AX
    idiv WORD PTR [bp-6]
    pop si                  ; Восстановить регистр SI
    mov sp, bp              ; Восстановить регистр SP
    pop bp                  ; Восстановить регистр BP
    retn                    ; Выйти из процедуры
_average ENDP
_TEXT   ENDS
        END

```

Так как для вызова процедуры `average` используются соглашения о связях Си, то передача ей переменного числа аргументов правомерна. Последним аргументом в списке аргументов переменной длины должен быть признак конца списка (в данном случае -1). Верхняя часть стека в момент первого вызова процедуры `average` будет выглядеть как на рис. 21.


```

        mov dx, sum_high          ; Вычислить среднее число и
        mov ax, sum_low           ; вернуть его в регистре AX
        idiv count
        ret                       ; Выйти из процедуры
average ENDP
END

```

Пример 2 представляет программу вывода на экран строки Си. Для вывода на экран строки Си Си-программа вызывает ассемблерную процедуру puts, используя соглашения о связях в стиле Паскаля.

Пример 2.

Программа на Си:

```

char near str[] = "Вывод в десятую строку с первой позиции";
main()
{
/* Объявление ассемблерной процедуры с использованием
соглашений о связях и именах Паскаля */
extern void pascal near puts(int, int, char near *);
int row, col;      /* Строка и столбец */
row = 9;           /* Вывод в десятую строку, начиная с */
col = 0;           /* первой позиции */
puts(row, col, str); /* Вызов процедуры near в стиле */
                    /* Паскаля */
}

```

Ассемблерная процедура:

```

_TEXT SEGMENT WORD PUBLIC 'CODE'
        ASSUME cs:_TEXT
        PUBLIC PUTS
PUTS    PROC
        push bp          ; Сохранить BP из Си-программы
        mov bp, sp       ; Зафиксировать указатель вершины
                           ; стека
        push si          ; Сохранить SI
        xor ax, ax
        xor bx, bx
        mov dh, BYTE PTR [bp + 8]; В DH — номер строки
        mov dl, BYTE PTR [bp + 6]; В DL — номер столбца
        mov si, WORD PTR [bp + 4]; В SI — адрес (near)
                           ; строки Си
        mov cx, 1H       ; В CX — число копий символа
        mov ah, 8H       ; Получить текущий атрибут символа
        int 10H
        mov bl, ah        ; Сохранить его в BL
write:  ; Цикл, пока не встретится нулевой
        ; символ конца строки Си
        mov ah, 2H       ; Установить курсор
        int 10H
        mov al, BYTE PTR [si]; Получить в AL текущий
        ; символ
        cmp al, 0H       ; Нулевой символ — выйти из цикла
        je exit

```



```
mov ah, 9H      ; Вывести текущий символ на экран
int 10H
inc si          ; Установить SI на следующий символ
inc di          ; Увеличить номер позиции в строке
                ; экрана
jmp write
exit:
pop si          ; Восстановить регистр SI
mov sp, bp      ; Восстановить указатель вершины
                ; стека
pop bp          ; Восстановить регистр BP
ret 6           ; Выйти из процедуры
PUTS           ENDP
_TEXT          ENDS
END
```

Так как для вызова процедуры puts используются соглашения о связях в стиле Паскаля, аргументы в процедуру puts передаются в том же порядке, в котором они заданы. Верхняя часть стека на входе в процедуру puts имеет вид, представленный на рис. 22.

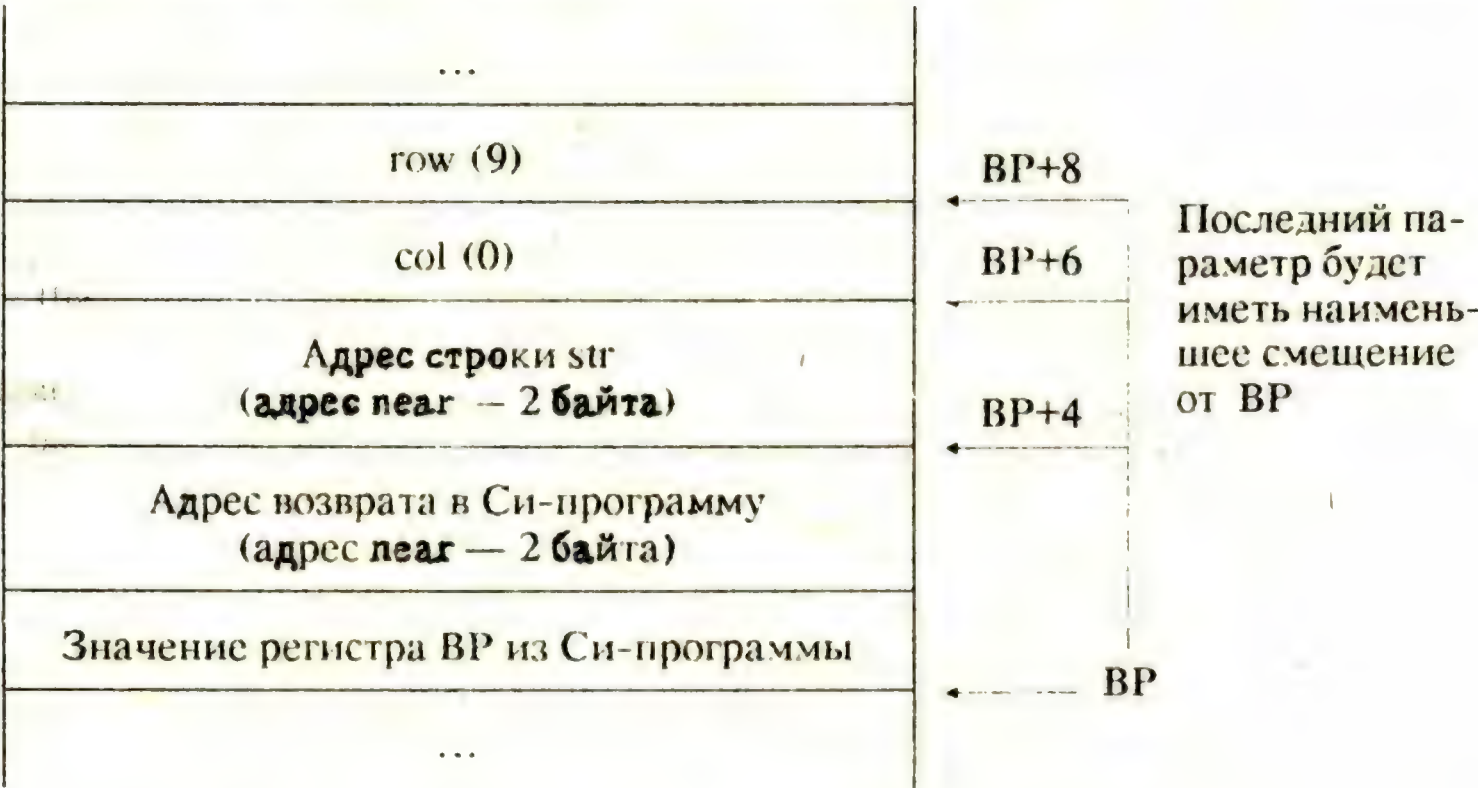


Рис. 22. Вид стека при вызове процедуры puts

Возвращаться из процедуры puts нужно по команде
ret 6

где 6 — суммарный размер передаваемых параметров.

Имя процедуры должно быть без символа подчеркивания в начале имени и представлено прописными буквами. Строка (массив символов) str в Си-программе размещается в сегменте данных по умолчанию (сегменте данных с именем _DATA), который адресуется сегментным регистром DS. В процедуру puts передается только смещение

строки `str` в сегменте данных. Регистр `DS` не изменяется, поэтому команда

```
mov al, BYTE PTR [si]
```

осуществляет правильный доступ к строке `str` (в команде подразумевается сегментный регистр `DS`).

Упрощенный вариант процедуры puts:

```
.MODEL SMALL,PASCAL
.CODE
PUTS PROC NEAR USES si,row:WORD,col:WORD,str:NEAR PTR BYTE
    xor ax, ax
    xor bx, bx
    mov dh, row           ; В DH — номер строки
    mov dl, col           ; В DL — номер столбца
    mov si, str           ; В SI — адрес (near) строки Си
    mov cx, 1H            ; В CX — число копий символа
    mov ah, 8H            ; Получить текущий атрибут символа
    int 10H
    mov bl, ah            ; Сохранить его в BL
write:                   ; Цикл пока не встретится нулевой
                           ; символ конца строки Си
    mov ah, 2H            ; Установить курсор
    int 10H
    mov al, BYTE PTR [si] ; Получить в AL текущий
                           ; символ
    cmp al, 0H            ; Нулевой символ — выйти из цикла
    je exit
    mov ah, 9H            ; Вывести текущий символ на экран
    int 10H
    inc si                ; Установить SI на следующий символ
    inc dl                ; Увеличить номер позиции в строке
                           ; экрана
    jmp write
exit:
    ret                  ; Выйти из процедуры
PUTS ENDP
END
```

10.5. ОРГАНИЗАЦИЯ СВЯЗИ С ПРОГРАММАМИ НА ПАСКАЛЕ И ФОРТРАНЕ

Языки Паскаль и Фортран используют одни и те же соглашения о связях и именах, поэтому интерфейс Си-программы с программами на этих языках одинаков для обоих языков и организуется по следующим правилам.

1. В Си-программе нужно объявить вызываемую программу как внешнюю функцию Си с одним из ключевых слов `pascal` или `fortran`. Для вызова программы через указатель надо объявить указатель на внешнюю функцию с одним из ключевых слов `pascal` или `fortran`.

Ключевые слова `pascal` и `fortran` интерпретируются следующим образом:

ключевые слова `pascal` и `fortran` модифицируют элемент, расположенный справа от ключевого слова;

ключевые слова `pascal` и `fortran` могут использоваться с ключевыми словами `near` и `far`, причем порядок следования не имеет значения. Например, запись `pascal far` эквивалентна записи `far pascal`.

Ключевые слова `pascal` и `fortran` взаимозаменяемы. Ключевое слово `pascal` можно использовать для объявления программы Фортрана и наоборот.

Примеры.

```
extern void pascal sort();
```

Объявляется процедура Паскаля или подпрограмма Фортрана. На то, что нет возвращаемого значения, указывает ключевое слово `void`.

```
extern int fortran far fact(int);
```

Объявляется функция Паскаля или Фортрана, которая возвращает значение целого типа.

```
extern int fortran fact();  
int (fortran * fun_ptr) ();
```

Объявляется указатель на функцию Паскаля или Фортрана, возвращающую целое значение.

2. Если в вызываемую программу передаются параметры, то в объявление необходимо включить список объявлений передаваемых параметров.

Примеры.

```
extern int pascal fact (int);
```

Объявляется функция Паскаля или Фортрана, которой передается аргумент типа `int` и которая возвращает значение типа `int`.

```
extern void fortran sort (int *, int *);
```

Объявляется процедура Паскаля или подпрограмма Фортрана. Передаются два указателя на значение целого типа. Размер указателей зависит от модели памяти: для средней модели это указатели `near`, а для большой модели — указатели `far`.

```
extern void fortran sort (int near *, int near *);
```

Аналогично предыдущему объявлению, но только в программу `sort` передаются два указателя `near`.

3. Аргументы в вызываемую программу передаются по значению, за исключением массивов, которые передаются ссылкой. Для передачи аргумента ссылкой нужно передать указатель на этот объект.

4. Вызвать программу Паскаля или Фортрана как обычную функ-

цию Си. Например, для вызова функции Паскаля следует выполнить следующую последовательность операторов:

```
extern int pascal fact (int);
int factorial, n;
...
n = 5;
factorial = fact(n);
```

5. При компиляции Си-программы, которая осуществляет интерфейс с программой Паскаля или Фортрана, должна быть задана средняя или большая модель памяти в соответствии с возможностями компиляторов Фортрана и Паскаля.

Вызов процедуры Паскаля

Процедура Паскаля в Си-программе объявляется как функция Си, не возвращающая значения.

Следующий пример демонстрирует вызов из Си-программы процедуры `gcd` Паскаля. Процедура `gcd` реализует алгоритм Евклида нахождения наибольшего общего делителя (НОД) двух положительных целых чисел.

Пример.

Программа на Си:

```
/* Эта программа вызывает процедуру Паскаля.
 * Программа должна быть откомпилирована со средней или
 * большой моделью памяти
 */
#include <stdio.h>
#include <stdlib.h>
/* Процедура gcd должна быть объявлена как функция Си с
 * ключевым словом pascal
 */
extern void pascal gcd(int, int, int near *);
                                1 3
main( int argc, char * argv[])
{
    int a, b, c;
    a = atoi(argv[1]);
    b = atoi(argv[2]);
    gcd(a, b, &c); /* Вычислить наибольший общий делитель */
                  /* чисел a и b и сохранить его в c */
    printf("Наибольший общий делитель чисел %d и %d равен\n", a, b, c);
}
```

Процедура на Паскале:

```
{ Процедура gcd вычисляет наибольший общий делитель двух }
{ целых чисел, которые передаются процедуре как первые   }
{ два параметра, и сохраняет его в третьем параметре,      }
{ который передается в виде указателя                       }
```



```

module P;
procedure gcd (x : integerc; y : integerc; var z : integerc) [public];
           5                2                4
    var t1, t2, t3 : integerc;
    begin
        t1 := x;
        t2 := y;
        while t2 > 0 do
            begin
                t3 := t1 mod t2;
                t1 := t2;
                t2 := t3;
            end;
        z := t1;
    end;
end.

```

Пояснения к примеру.

1 — так как процедура gcd сохраняет значение наибольшего общего делителя в переменной "с" из функции Си main, то процедуре эту переменную нужно передать ссылкой, а не по значению. Передача переменной "с" ссылкой — это передача адреса этой переменной.

2 — процедуре gcd адрес переменной "с" передается как указатель near. Поэтому в заголовке процедуры gcd следует использовать ключевое слово Паскаля var. Если бы адрес передавался в виде указателя far, то принимаемый параметр в заголовке процедуры gcd требовалось бы объявить с ключевым словом vars.

3 — при компиляции функции main нужно указать среднюю или большую модель памяти. Если бы в объявлении указателя, передаваемого процедуре gcd, отсутствовало ключевое слово near, то размер указателя зависел бы от модели памяти. Так как это указатель на данные, то для средней модели памяти передавался бы указатель near, а для большой модели — указатель far. Отсутствие явной спецификации размера передаваемых указателей является потенциальной причиной возможных неожиданных результатов при выполнении этой программы. Наличие ключевого слова near в этой программе гарантирует ее правильное выполнение.

4 — процедура gcd определена с атрибутом public, означающим, что процедура gcd является глобальной. Исходный файл Паскаля, содержащий процедуру gcd, оформлен в виде модуля, поэтому атрибут PUBLIC можно опустить. По умолчанию в Паскале все процедуры и функции модуля являются глобальными.

5 — целый тип Паскаля integerc является совместимым с типом int Си.

Вызов функции Паскаля

Функция Паскаля в Си-программе объявляется как функция Си, возвращающая значение.

Следующий пример демонстрирует вызов из Си-программы функции `fact` Паскаля. Функция `fact` вычисляет факториал числа, передаваемого из функции Си `main`. Функция `main` печатает значение, возвращаемое `fact`.

Пример.

Программа на Си:

```
/* Эта программа вызывает функцию Паскаля.
 * Программа должна быть откомпилирована со средней или
 * большой моделью памяти
 */
#include <stdio.h>
/* Функция fact должна быть объявлена с ключевым словом
 * pascal. Это ключевое слово означает, что для вызова
 * функции fact будут использоваться соглашения о связях
 * и именах Паскаля
 */
extern int pascal fact(int);
main()
{
    int x;
    x = 5;
    /* Вычислить факториал числа и напечатать результат */
    /* Параметр x передается по значению */
    printf("Факториал числа %d равен %d\n", x, fact(x));
}
```

Функция Паскаля:

```
{ Функция fact вычисляет факториал числа, которое передает- }
{ ся ей как параметр }
module Pfun;
{ Целые величины передаются по значению }
function Fact (n : integerc) : integerc [public];
begin
    Fact := 1;
    if n > 1 then
        Fact := Fact(n - 1) * n;
    end;
end.
```

Аргументы в функцию Паскаля `fact` передаются по значению, так как по умолчанию в Си и Паскале аргументы передаются по значению.

Вызов подпрограммы Фортрана

Подпрограмма Фортрана в Си-программе объявляется как функция Си, не возвращающая значений.

Следующий пример демонстрирует вызов из Си-программы под-

программы Фортрана с именем gcd. Подпрограмма gcd выполняет то же, что и процедура gcd Паскаля из предыдущего пункта.

Пример.

Программа на Си:

```
/* Эта программа вызывает подпрограмму Фортрана.
 * Программа должна быть откомпилирована со средней или
 * большой моделью памяти
 */
#include <stdio.h>
#include <stdlib.h>
extern void fortran gcd (int, int, int near *);
                                1 2
main (int argc, char * argv[])
{
    int a, b, c;
    a = atoi (argv[1]);
    b = atoi (argv[2]);
    gcd (a, b, &c); /* Вычислить наибольший общий делитель */
                  /* чисел a и b и сохранить его в c */
    printf ("Наибольший общий делитель чисел %d и %d равен
            %d\n", a, b, c);
}
```

Подпрограмма на Фортране:

С Подпрограмма gcd вычисляет наибольший общий делитель
 С двух целых чисел, которые передаются подпрограмме
 С как первые два параметра, и сохраняет его в третьем
 С параметре, который передается ссылкой
 С

```

SUBROUTINE GCD(X, Y, Z)
INTEGER[C] X [VALUE]           3 4
INTEGER[C] Y [VALUE]
INTEGER[C] Z [NEAR]           2
INTEGER[C] T1, T2, T3
T1 = X
T2 = Y
10  T3 = MOD (T1, T2)
    IF (T3 .EQ. 0) GOTO 20
    T1 = T2
    T2 = T3
    GOTO 10
20  Z = T1
    END
```

Пояснения к примеру.

1 — так как подпрограмма gcd изменяет значение переменной "с" из вызывающей программы, то эта переменная должна быть передана подпрограмме ссылкой.

2 — адрес переменной "с" в подпрограмму gcd передается как указатель near, на что указывает ключевое слово near в объявлении типа

передаваемого параметра. Поэтому в заголовке подпрограммы gcd передаваемый ей параметр должен быть объявлен с атрибутом Фортрана NEAR.

Если в Си-программе параметр, передаваемый подпрограмме gcd, был бы объявлен как указатель far, то принимаемый параметр в заголовке подпрограммы gcd следовало бы объявлять с атрибутом FAR.

3 — целый тип Фортрана INTEGER с атрибутом C является совместимым с типом int Си.

4 — атрибут VALUE Фортрана означает принятие передаваемого параметра по значению.

Вызов функции Фортрана

Функция Фортрана объявляется в Си-программе как функция Си, возвращающая значение.

Следующий пример демонстрирует, как Си-программа для вычисления факториала вызывает функцию Фортрана и печатает полученный результат.

Пример.

Программа на Си:

```
/* Эта программа вызывает функцию Фортрана.
 * Программа должна быть откомпилирована со средней или
 * большой моделью памяти
 */
#include <stdio.h>
/* Функция fact должна быть объявлена с ключевым словом
 * fortran, которое означает, что для вызова функции fact
 * будут использоваться соглашения о связях и именах
 * Фортрана
 */
extern int fortran fact(int);
main()
{
    int x;
    x = 6;
    /* Вычислить факториал числа и напечатать результат */
    /* Параметр x передается по значению */
    printf("Факториал числа %d равен %d\n", x, fact(x));
}
```

Функция на Фортране:

С Функция fact вычисляет факториал числа, которое
С передается ей как параметр

```
INTEGER[C] FUNCTION FACT (N)
INTEGER[C] N [VALUE]
```

С
С
С
С

Целое число принимается по значению, так как
задан атрибут VALUE


```

      INTEGER[C] I
      FACT = 1
      DO 10 I = 1, N
        FACT = FACT * I
10     CONTINUE
      RETURN
      END

```

Параметр N в заголовке функции fact объявлен с атрибутом VALUE. Атрибут Фортрана VALUE означает, что аргумент x в функцию fact передается по значению. Атрибут VALUE используется, чтобы изменить принятый по умолчанию в Фортране способ передачи параметров ссылкой.

Альтернативная форма связи с программами на Паскале и Фортране

Возможна альтернативная форма связи с программами на Паскале и Фортране. Эта форма связи заключается в том, что для связи с программами на Паскале и Фортране используются соглашения о связях и именах Си. Чтобы использовать эту форму связи, вызываемая программа Паскаля или Фортрана должна быть определена с атрибутом C в заголовке программы. Атрибут C Паскаля или Фортрана означает, что вызываемая программа будет оформлена в соответствии с соглашениями о связях и именах Си, а именно:

- 1) параметры, передаваемые программе, будут приниматься в том же порядке, как и в функции Си;
- 2) вызываемая программа не будет содержать команд восстановления стека;
- 3) параметры, передаваемые программе, принимаются по значению; массивы передаются ссылкой;
- 4) внешние имена будут содержать только строчные буквы и начинаться с символа подчеркивания.

Заголовок функции Паскаля с атрибутом C:

```

module pfun;
function Fact (n:integer):integer [c];

```

Заголовок функции Фортрана с атрибутом C:

```

INTEGER*2 FUNCTION FACT [C] (N)
INTEGER*2  N

```

Так как программа Паскаля или Фортрана с атрибутом C будет оформлена в соответствии с соглашениями о связях и именах Си, то в Си-программе эта программа должна быть объявлена без ключевого слова pascal или fortran.

Для того чтобы подтвердить использование соглашений Си для связи с программами на Паскале или Фортране, в объявлении этих про-

грамм в Си-программе можно использовать ключевое слово `cdecl`. Ключевое слово `cdecl` обязательно нужно указать для тех программ на Паскале или Фортране, которые используют для связи соглашения Си, если для компилятора Си был установлен режим использования соглашений Паскаля (Фортрана), распространяющийся на все функции программы.

Перепишем ранее рассмотренную программу вычисления факториала числа с использованием соглашений о связях Си.

Вариант Си-программы:

```
/* Эта программа вызывает функцию fact, которая может
 * быть написана как на Паскале, так и на Фортране.
 * Программа должна быть откомпилирована со средней или
 * большой моделью памяти
 */
#include <stdio.h>
/* Функция fact должна быть объявлена без ключевого слова
 * pascal, так как функция fact определена с атрибутом C,
 * означающим, что будут использоваться соглашения о связях
 * и именах Си
 */
extern int fact(int);
main()
{
    int x;
    x = 5;
    /* Вычислить факториал числа и напечатать результат */
    /* Параметр x передается по значению */
    printf("Факториал числа %d равен %d\n", x, fact(x));
}
```

Функция на Паскале (вызов в стиле Си):

```
{ Функция fact вычисляет факториал числа, которое переда-
{ ется ей как параметр }
module Pfun;
{ Атрибут C означает, что для вызова функции fact исполь- }
{ зуются соглашения о связях и именах Си. Целые величины }
{ передаются по значению }
function Fact (n : integer) : integer [c];
begin
    Fact := 1;
    if n > 1 then
        Fact := Fact(n - 1) * n;
    end;
end.
```

Функция на Фортране (вызов в стиле Си):

```
C Функция fact вычисляет факториал числа, которое переда-
C ется функции как параметр. Функция fact определена с
C атрибутом C, означающим, что используются соглашения о
C связях и именах Си
      INTEGER*2 FUNCTION FACT [C] (N)
```



```
INTEGER*2 N
C
C Целое число передается по значению
C
      INTEGER*2 I
      FACT = 1 DO 10 I = 1, N
        FACT = FACT * I
10    CONTINUE
      RETURN
      END
```

Заметим, что параметр N в функции FACT объявлен без атрибута VALUE, означающего передачу параметров по значению. Атрибут C в заголовке функции FACT определяет принятие целых величин по значению.

Элементы языков Си, Паскаль и Фортран, определяющие форму соглашений о связях, сведены в табл. 10.5.

Таблица 10.5

Определение формы соглашений о связях

Язык	Определение связи в стиле Си	Определение связи в стиле Паскаля
Си (язык вызывающей программы)	Режим по умолчанию	Ключевые слова pascal и fortran в объявлении программы
Паскаль (язык вызываемой программы)	Атрибут C в заголовке процедуры/функции	Режим по умолчанию
Фортран (язык вызываемой программы)	Атрибут C в заголовке подпрограммы или функции или в операторе INTERFACE	Режим по умолчанию

Передача данных Си в программы на Паскале и Фортране

Язык Си определяет передачу аргументов в вызываемую программу как передачу данных по значению. Исключение составляют массивы, которые передаются ссылкой, т.е. при передаче массива передается адрес начала массива. Строки в Си являются массивами символов и поэтому также передаются ссылкой.

Массив можно передать по значению, если объявить его как элемент структуры, например,

```
struct S_ARR {
```



```
        int arr[10];  
    } pass;  
void pascal subpr (struct S_ARR);  
...  
subpr(pass);
```

В программе `subpr` структура `pass` принимается как массив из десяти элементов.

Большие элементы данных, такие как массивы и структуры, эффективнее передавать ссылкой, а не значением.

Передача целых и плавающих типов. При передаче данных целых и плавающих типов в программы на Паскале и Фортране необходимо учитывать следующее:

1. Данные типа `char` (`unsigned char`) преобразуются к типу `int` (`unsigned int`) перед передачей их по значению. Чтобы избежать этих преобразований, можно данное типа `char` (`unsigned char`) объявить как элемент структуры и в подпрограмму передать структуру, например,

```
struct S {  
    char c;  
    } x;  
void pascal proc (struct S);  
...  
proc (x);
```

2. Данные типа `float` преобразуются к типу `double`, если они передаются по значению и отсутствует прототип вызываемой программы с типами формальных параметров.

3. Для совместимости с типом `Si int` в Паскале введен тип `integerc`, а в Фортране используется тип `INTEGER` с атрибутом `C` (`INTEGER[C]`).

4. В языке Фортран отсутствуют целые типы без знака. Эти величины можно принимать как данные типов `INTEGER*1`, `INTEGER*2` или `INTEGER*4`, в зависимости от длины принимаемых параметров. Не следует только передавать величины, превышающие максимально допустимые значения для целых типов Фортрана (127 для типа `INTEGER*1`, 32767 для типа `INTEGER*2` и 2 147 483 647 для типа `INTEGER*4`).

5. В языке Фортран нет типа данных перечисление. Данные типа перечисление в языке `Си` рассматриваются как данные типа `int`, т.е. переменная типа перечисление имеет такой же размер, как и переменная типа `int`. Поэтому в программе на Фортране можно организовать принятие таких данных, как данные типа `INTEGER [C]` или `INTEGER*2`.

Передача строк. При передаче строк необходимо учитывать, что формат представления строк в языке Си отличается от представления строк в Паскале и Фортране. В языке Си строка рассматривается как массив символов. В конец каждой строки добавляется нулевой символ (символ с шестнадцатеричным кодом '00'), отмечающий конец строки. Следовательно, конец строки всегда известен. Размер строки Си на единицу превышает количество символов в строке.

Например, строка

Это строка Си

в языке Си определяется следующим образом:

```
char cstr[] = "Это строка Си";
```

В памяти такая строка занимает не тринадцать, а четырнадцать байтов:

Э	т	о		с	т	р	о	к	а		С	и	\0
---	---	---	--	---	---	---	---	---	---	--	---	---	----

Строки Си передаются ссылкой (так как это массив), т.е. передается не сама строка, а указатель на строку. Например,

```
int pascal fun(char *);
char str[] = "Это строка Си";
...
fun(str);
```

Можно передать и всю строку. Для этого нужно объявить структуру, состоящую из одного элемента — строки, и передать структуру. Например,

```
struct CSTR {
    char str[14];
} sstr;
int pascal fun (struct CSTR);
...
/* заполнение строки str */
...
fun (sstr);
```

Этот способ передачи строк не является эффективным и рассматривается скорее как теоретическая возможность.

Принимать строку Си в Паскале и Фортране следует как указатель на строку (массив символов) переменной длины.

Объявление принимаемой строки Си в программе на Паскале и Фортране показано в табл. 10.6.

Таблица 10.6

Объявление принимаемой строки Си
в программе на Паскале и Фортране

Объявление строки в Си	Объявление принимае- мой строки Си в Паскале	Объявление принимаемой строки Си в Фортране
char x[n]	var x: string type cstr=super array [1..*] of char var x:cstr	CHARACTER*1 X DIMENSION X(*)

Следующий пример показывает передачу строки Си в программу на Паскале и Фортране.

Для вычисления длины строки (длина строки Си не включает нулевой символ конца строки) функция Си main вызывает функцию Паскаля (Фортрана). Строка Си передается ссылкой.

Пример.

Вариант Си-программы с использованием функции Паскаля:

```
/* Эта программа демонстрирует передачу строки Си в функцию
 * Паскаля. Используются соглашения о связях Паскаля.
 */
#include <stdio.h>
main()
{
    int lstr;
    int pascal len(char near *);
    /* Вычислить и напечатать длину строки */
    lstr = len("Это строка Си"); /* Передается адрес строки */
    printf("Длина строки: %d\n", lstr);
}
```

Функция на Паскале:

```
{ Функция len вычисляет длину строки Си, которая передается }
{ в функцию как указатель на строку. В функции len строка }
{ Си обрабатывается как массив символов переменной длины }
module P;
{ Параметр str принимается как указатель на массив символов }
{ переменной длины }
function Len (var str : string) : integerc[public];
    var i, j : integerc;
    begin
        i := 1;
        j := 0;
        while str[i] <> chr(0) do
            begin
                i := i + 1;
                j := j + 1;
            end;
        Len := j;
    end;
end.
```


Вариант Си-программы с использованием функции Фортрана:

```

/* Эта программа демонстрирует передачу строки Си в
 * функцию Фортрана. Используются соглашения о связях
 * Фортрана.
 */
#include <stdio.h>
main()
{
    int lstr;
    int fortran len(char near *);
    /* Вычислить и напечатать длину строки */
    lstr = len("Это строка Си"); /*Передается адрес строки */
    printf("Длина строки: %d\n", lstr);
}

```

Функция на Фортране:

С Функция len вычисляет длину строки Си, которая передается

С функции ссылкой

```

INTEGER(C) FUNCTION LEN (CSTR)
CHARACTER CSTR [NEAR]
DIMENSION CSTR (*)

```

С Цикл, пока не встретится нулевой символ

```

INTEGER(C) I
I = 1
10  IF (CSTR(I) .EQ. 'C') GOTO 20
    I = I + 1
    GOTO 10
20  LEN = I - 1
    RETURN
END

```

Передача массивов. При передаче массивов следует учитывать, что массивы передаются ссылкой, т.е. передается адрес начала массива, и что в языках Си, Паскаль и Фортран массивы индексируются и хранятся по-разному:

1) в Си и Паскале индексация массивов ведется (по умолчанию) с 0, а в Фортране — с 1. В Паскале, однако, нижний индекс массива может быть любым целым числом;

2) в Си и Паскале массивы хранятся по строкам, а в Фортране — по столбцам. Например, массив Си `a[2][3]` будет храниться следующим образом:

`a[0][0] a[0][1] a[0][2] a[1][0] a[1][1] a[1][2]`

тогда как массив Фортрана `A(2,3)` запоминается следующим образом:

`A(1,1) A(2,1) A(1,2) A(2,2) A(1,3) A(2,3)`

Поэтому объявление массива в Си

`long a[2][3];`

эквивалентно следующему объявлению массива в Фортране

`INTEGER*4 A(3,2)`

Объявление принимаемых одномерных и многомерных массивов Си в программе на Паскале и Фортране показано в табл. 10.7.

Таблица 10.7

Объявление принимаемого массива Си
в программах на Паскале и Фортране

Объявление массива в Си	Объявление принимаемого массива в Паскале	Объявление принимаемого массива в Фортране
тип x[n]	type a = array[i..i+n-1] of тип var x:a	тип X DIMENSION X(n)
тип x[l][m][n]	type a = array [i..i+l-1, j..j+m-1, k..k+n-1] of тип var x:a	тип X DIMENSION X(n,m,l)

Следующий пример представляет программу сортировки по возрастанию массива целых чисел. Для сортировки массива, который передается ссылкой, вызывается процедура Паскаля (подпрограмма Фортрана).

Пример.

Вариант Си-программы с использованием процедуры Паскаля или подпрограммы Фортрана:

```
/* Эта программа демонстрирует передачу массива из Си-про-
 * граммы в процедуру Паскаля или подпрограмму Фортрана.
 * Массив передается ссылкой. Используются соглашения
 * о связях Паскаля (Фортрана)
 */
```

```
#include <stdio.h>
int arr[20] = { 12, 1, 3, 24, 56, 0, 1, 6, 7, 23,
               10, 9, 11, 25, 3, 5, 8, 15, 20, 33 };

main()
{
    int i;
    void pascal srt(int near *);
    printf("Массив до сортировки\n");
    for (i = 0; i < 20; i++)
        printf("%d%c", arr[i], (i == 19) ? '\n' : ' ');
    srt(arr);
    printf("Массив после сортировки\n");
    for (i = 0; i < 20; i++)
        printf("%d%c", arr[i], (i == 19) ? '\n' : ' ');
}
```

Процедура на Паскале:

```
{ Процедура Паскаля srt сортирует по возрастанию массив }
{ целых чисел, который передается из Си-программы ссылкой }
module P;
    type arr = array [1 .. 20] of integer;
```



```

procedure Srt (var a : arr) [public];
  var temp : integer;
begin
  for var i := 1 to 20 do
    for var j := i + 1 to 20 do
      if a[i] > a[j] then
        begin
          temp := a[i];
          a[i] := a[j];
          a[j] := temp;
        end;
      end;
    end;
  end;
end.

```

Подпрограмма на Фортране:

С Подпрограмма Фортрана srt сортирует по возрастанию массив
 С целых чисел, который передается ей из Си-программы

```

SUBROUTINE SRT(A)
  INTEGER(C) A [NEAR]
  DIMENSION A(20)
  С Массив передается в виде ссылки near (задан атрибут NEAR)
  INTEGER*2 I, J
  INTEGER(C) TEMP
  DO 20 I = 1, 20
    DO 10 J = I+1, 20
      IF (A(I) .GT. A(J)) THEN
        TEMP = A(I)
        A(I) = A(J)
        A(J) = TEMP
      ENDIF
    CONTINUE
  CONTINUE
END

```


ИСПОЛЬЗОВАНИЕ ФУНКЦИЙ БИБЛИОТЕКИ ПРИ ПРОГРАММИРОВАНИИ ПРИМЕНЕНИЙ

Глава 11

ВВОД И ВЫВОД

В языке Си нет встроенных в язык средств ввода и вывода. Все действия, связанные с вводом-выводом, выполняются с помощью функций библиотеки Си. Программист может использовать эти функции или разрабатывать собственные средства ввода-вывода и включать их в библиотеку.

Си не предусматривает никаких predetermined структур файлов: файлы рассматриваются как последовательности байтов.

Функции ввода и вывода библиотеки Си позволяют читать данные из файлов и с устройств и писать данные в файлы и на устройства. Библиотека Си поддерживает следующие три уровня ввода-вывода: ввод-вывод потока, ввод-вывод нижнего уровня и ввод-вывод для консоли и порта.

11.1. ВВОД-ВЫВОД ПОТОКА

При вводе-выводе потока все данные рассматриваются как поток отдельных байтов. Для пользователя *поток* — это либо файл на диске, либо физическое устройство, например, такое, как дисплей или печатающее устройство. Поэтому, когда пользователь выполняет операции ввода-вывода для потока, то имеется в виду, что он работает либо с файлами, либо с устройством. Хотя поток есть последовательность отдельных байтов, функции ввода-вывода для потока позволяют обрабатывать данные различных размеров и форматов (от одиночного символа до больших структур данных), обеспечивая при этом буферизованный форматизованный или неформатизованный ввод и вывод.

Ввод-вывод потока позволяет:

- открывать и закрывать потоки;

- создавать и удалять временные потоки;

- читать и записывать символ;

- читать и записывать строки;

- читать и записывать форматизованные данные;

- читать и записывать неформатизованные данные;

анализировать ошибки ввода-вывода потока и условия конца потока (конца файла);

управлять буферизацией потока и размером буфера;

выгружать буфера, связанные с потоками;

получать и устанавливать указатель текущей позиции в потоке.

Полный список функций ввода-вывода потока и их прототипы приведены в приложении 2.

Чтобы использовать функции ввода-вывода потока в программе, необходимо директивой `#include` включить файл `stdio.h`. Этот файл содержит объявления функций ввода-вывода, а также определения констант, типов и структур, используемых функциями потока.

Открытие потока

Перед выполнением операций ввода-вывода для потока его нужно открыть. Когда поток открывается для операций ввода-вывода, он связывается со структурой предопределенного типа `FILE`, содержащей всю необходимую информацию для работы с потоком: такую, как указатель текущей позиции в потоке, указатель на буфер, связанный с потоком, тип доступа к потоку и др. При открытии потока возвращается указатель на структуру `FILE`, называемый указателем потока. Этот указатель используется для последующих ссылок на поток.

Для открытия потока библиотека Си предоставляет три функции: `fopen` и `fdopen` открывают поток, а `freopen` переназначает указатель потока на другой поток.

При успешном открытии потока функции открытия возвращают указатель на структуру `FILE`, в противном случае возвращается нулевой указатель (`NULL`). Возвращаемое значение указателя `FILE` необходимо присвоить переменной и использовать эту переменную для ссылки на открытый поток. Ниже приведена стандартная последовательность операторов, необходимая для открытия потока:

```
#include <stdio.h>
...
FILE *file_ptr;
...
if ((file_ptr=fopen(...)) == NULL)
    printf("Ошибка открытия ... \n");
```

Функции открытия потока позволяют открыть поток для операций чтения, операций записи или операций чтения и записи. Кроме того, поток может быть открыт в текстовом или в двоичном режиме. Тип доступа к потоку задается как аргумент функции открытия. Он явля-

ется строковым литералом, который может принимать следующие значения:

"r" — поток открывается для чтения;

"w" — открывается пустой поток для записи. Если поток существует, его содержимое пропадает;

"a" — поток открывается для записи в конец потока. Если поток не существует, он создается;

"r+" — поток открывается для чтения и записи (поток должен существовать);

"w+" — открывается пустой поток для чтения и записи. Если поток существует, его содержимое пропадает;

"a+" — поток открывается для чтения и записи в конец потока. Если поток не существует, он создается.

Следующий пример показывает открытие потока, связанного с файлом на диске.

Пример.

```
#include <stdio.h> /* Требуется для объявлений функций */
```

```
...
```

```
FILE *file_ptr;
```

```
...
```

```
if((file_ptr=fopen("a:file1.dat","r")) == NULL)
```

```
    printf("Ошибка открытия файла \n");
```

```
else
```

```
    printf("Файл открыт для чтения \n");
```

Предопределенные указатели потоков

С началом выполнения Си-программы автоматически открывается пять потоков. Этими потоками являются: стандартный ввод, стандартный вывод, стандартный вывод сообщений об ошибках, стандартный дополнительный поток и стандартная печать. По умолчанию стандартные ввод, вывод и вывод сообщений об ошибках относятся к консоли пользователя. Это означает, что всякий раз, когда программа ожидает ввод со стандартного ввода, данные должны поступать с консоли. Аналогично, когда программа пишет в стандартный вывод, данные выводятся на консоль. Сообщения об ошибках, посылаемые в стандартный вывод сообщений об ошибках, также выводятся на консоль пользователя.

Назначения стандартному дополнительному потоку и стандартной печати зависят от конфигурации машины. Обычно стандартный дополнительный поток относится к дополнительному порту, к которому, например, может быть подключена вторая консоль, а стандартная печать — к печатающему устройству.

На пять стандартных потоков можно ссылаться, используя следующие predefined указатели потоков:

stdin — стандартный ввод;
 stdout — стандартный вывод;
 stderr — стандартный вывод сообщений об ошибках;
 stderr — стандартный дополнительный поток;
 stderr — стандартная печать.

Эти указатели можно использовать в любой функции ввода-вывода, которая в качестве аргумента требует указатель потока. Следующий пример 1 представляет функцию открытия файла, которая выдает сообщение об ошибке открытия в стандартный поток stderr, если необходимо.

Пример 1.

```
/* Функция открытия файла с выдачей диагностического
 * сообщения в случае ошибки открытия
 */
#include <stdio.h>      /* Требуется для объявления функций */
FILE * open_file(path, type) /* Определение в старом стиле */
char * path;      /* Открываемый файл */
char * type;      /* Тип доступа к файлу */
{
    FILE * fp;
    /* Файл открыт успешно ? */
    if ((fp = fopen(path, type)) == NULL) {
        /* Нет — печатать сообщение об ошибке */
        unsigned char mode;
        char c;
        while (c = *type++)
            switch (c) {
                case 'r':    mode = 1;
                             break;
                case 'w':    mode = 2;
                             break;
                case 'a':    mode = 4;
                             break;
                case '+':    mode |= 3;
                             break;
            }
        fprintf(stderr, "Нельзя открыть файл \"%s\" для \"%s\n\", path,
            (mode == 1) ? "чтения" :
            ((mode == 2) ? "записи" :
            ((mode == 3) ? "чтения и записи" :
            ((mode == 4) ? "добавления" : "чтения и добавления"))));
    }
    return(fp);
}
```

Указатели stdin, stdout, stderr, stderr и stderr являются константами, а не переменными, поэтому им нельзя присваивать новое значение.

ние указателя потока. Однако любой из этих потоков можно с помощью функции freopen переадресовать на другое устройство или в другой поток, связанный с файлом на диске, например,

```
freopen ("file1.dat", "w", stdout);
```

Этот оператор закрывает поток для стандартного вывода (если только до этого stdout не был переадресован на другой поток) и пере-назначает stdout файлу file1.dat. После выполнения этого оператора любой вывод в stdout означает вывод в файл file1.dat.

Чтение из потока и запись в поток

Функции потока позволяют осуществлять ввод-вывод данных различными способами. Можно читать и записывать данные как последовательность байтов (двоичные данные) или задавать чтение и запись символов, строк или данных более сложных форматов.

Классификация функций чтения и записи по типу объекта, над которым выполняется операция, и разновидности потока приведены в табл. 11.1.

Таблица 11.1

Классификация функций чтения и записи потока

Объект опера- ции	Операция ввода-вывода					
	Чтение			Запись		
	из потока stdin	из любого потока	из строки Си	в поток stdout	в любой поток	в строку Си
После- дова- тель- ность байтов		fread			fwrite	
Отдель- ный символ	getc getchar	fgetc fgetchar		putc putchar ungetc	fputc fputchar	
Данное типа int		getw			putw	
Строка	gets	fgets		puts	fputs	
Форма- тизо- ванные данные	scanf	fscanf	sscanf	printf vprintf	fprintf vfprintf	sprintf vsprintf

Операции чтения и записи для потока начинаются с текущей позиции потока, определяемой указателем потока. Начальная позиция указателя потока устанавливается при открытии потока и может указывать на начало или на конец потока в зависимости от того, с каким типом доступа открыт поток (типы "r" и "w" устанавливают поток на начало, а тип "a" — на конец). После выполнения операции чтения-записи указатель потока изменяется для того, чтобы отразить новую текущую позицию потока. Например, если из потока был прочитан один символ, указатель потока увеличивается на единицу, чтобы следующая операция начиналась с первого байта после прочитанного.

Указатель потока можно позиционировать на любое место в потоке. Следующая операция ввода-вывода будет выполняться с указанной позиции, т.е. позиционирование позволяет осуществлять прямой ввод-вывод для потока. Библиотека функций Си предоставляет пять функций для позиционирования указателя потока:

`ftell` и `fgetpos` — получают текущую позицию указателя потока;

`fseek` и `fsetpos` — устанавливают текущую позицию указателя потока;

`rewind` — позиционирует указатель потока на начало потока.

Новая позиция указателя потока в функции `fseek` задается как смещение от некоторой начальной позиции, которая может быть одним из следующих значений:

`SEEK_SET` — начало потока,

`SEEK_CUR` — текущая позиция указателя потока,

`SEEK_END` — конец потока.

Примеры использования функции `fseek`:

`FILE *fp;`

`long n;`

...

`fseek(fp, 0L, SEEK_SET);` /* Позиционирование на начало потока */

`fseek(fp, 0L, SEEK_END);` /* Позиционирование на конец потока */

`fseek(fp, n, SEEK_SET);` /* Позиционирование на n байт от
начала потока */

`fseek(fp, n, SEEK_CUR);` /* Позиционирование на n байт от
текущей позиции */

`fseek(fp, -n, SEEK_CUR);` /* Позиционирование на n байт до
текущей позиции */

`fseek(fp, -n, SEEK_END);` /* Позиционирование на n байт до
конца потока */


```
fseek(fp,n,SEEK_END); /* Указатель потока может быть
                        позиционирован за конец потока */
fseek(fp,-n,SEEK_SET); /* Попытка установить указатель потока
                        перед началом потока приведет к
                        ошибке */
```

Функция `rewind(fp)` эквивалентна функции `fseek(fp, 0L, SEEK_SET)`. Функция `ftell` возвращает указатель текущей позиции потока.

Функции `fgetpos` и `fsetpos` обычно используются в паре. Функция `fgetpos` получает текущее значение указателя потока и сохраняет его в некоторой переменной, а функция `fsetpos` устанавливает указатель потока на значение, полученное функцией `fgetpos`.

Примеры 2 и 3 выполняют одинаковое действие, но различными средствами. Функция примера 2 читает группу байтов с заданной позиции потока и возвращает число прочитанных байтов. Функция примера 3 читает группу байтов с использованием функций `fgetpos` и `fsetpos`. Позиция указателя потока не меняется.

Пример 2.

```
/* Чтение n байт из потока fp с позиции pos */
#include <stdio.h> /* Требуется для объявления функций */
long random_read_bytes(fp, pos, n, save)
FILE * fp;        /* Указатель на файл для чтения */
long pos,         /* Позиция для чтения */
    n;           /* Количество байтов для чтения */
char * save;      /* Буфер для чтения */
{
    long num = -1L; /* Число прочитанных байтов */
    if (fseek(fp, pos, SEEK_SET) == 0)
        num = fread(save, 1, n, fp);
    return num;
}
```

Пример 3.

```
/* Чтение n байт из потока fp с текущей позиции */
#include <stdio.h> /* Требуется для объявления функций */
long random_read_bytes(fp, n, save)
FILE * fp;        /* Указатель на файл для чтения */
long n;           /* Количество байтов для чтения */
char * save;      /* Буфер для чтения */
{
    long num,      /* Число прочитанных байтов */
        pos;      /* Текущая позиция файла */
    num = -1L;
```



```
if (fgetpos(fp, &pos) == 0) /* Сохранить текущую позицию
                             файла */
    num = fread(save, 1, n, fp); /* Прочитать n байт */
    fsetpos(fp, &pos); /* Восстановить указатель текущей
                        позиции */
}
return num;
}
```

Если поток открывается для добавления (тип доступа "а" или "а+"), операции записи относятся к концу потока. Хотя указатель потока может быть переустановлен, он всегда позиционируется на конец потока перед выполнением операции записи. Это предохраняет данные потока от затирания.

Нельзя использовать функции определения и установки указателя потока для потока, представляющего устройство. Использование функций `fseek`, `fgetpos`, `fsetpos`, `ftell` и `rewind` с любым из стандартных потоков приводит к неопределенным результатам.

Заккрытие потока

Для закрытия потока или потоков используются функции `fclose` и `fcloseall`. Функция `fclose` закрывает отдельный указанный поток; функция `fcloseall` закрывает все открытые потоки, за исключением `stdin`, `stdout`, `stderr`, `stdaux` и `stdprn`. Все буфера, связанные с потоком, перед закрытием потока выгружаются. Если поток явно не закрывается в программе, он автоматически закрывается при завершении программы. Однако лучше закрывать поток, если с ним закончена работа, так как количество одновременно открытых потоков в операционной системе Альфа-ДОС ограничено. Система программирования Си разрешает иметь до 20 одновременно открытых потоков.

Обработка ошибок

При вызове функций ввода-вывода потока полезно анализировать возвращаемое функцией значение и обрабатывать ошибочные ситуации, если возвращаемое значение указывает на ошибку. Иначе в результате работы программы могут быть получены непредсказуемые результаты.

Значения, возвращаемые функциями ввода-вывода потока в случае успешного и неуспешного их завершения, приведены в табл. 11.2.

Функции чтения из потока возвращают значение EOF (или NULL для функций чтения строк) при возникновении условия конца потока или ошибки чтения. Поэтому для этих функций проверка на ошибку или на конец потока должна выполняться с помощью функции `ferror` или `feof`.

Таблица 11.2

Значения возврата функций ввода-вывода потока

Функция	Возвращаемое значение в случае	
	успешного завершения	неуспешного завершения
Открытие потока fopen fdopen freopen	Указатель на поток	NULL
Закрытие потока fclose fcloseall	0 Число закрытых потоков	EOF
Чтение из потока последовательности байтов fread	Число прочитанных элементов	Число прочитанных эле- ментов меньше, чем за- прашивалось
отдельного символа fgetc fgetchar getc getchar	Прочитанный символ	EOF
данного типа int getw	Прочитанное целое значе- ние	EOF
строки fgets gets	Указатель на прочитан- ную строку	NULL
форматизованных данных семейство функций scanf	Число успешно прочитан- ных полей	EOF
Запись в поток последовательности байтов fwrite	Число записанных эле- ментов	Число записанных эле- ментов меньше, чем за- прашивалось
отдельного символа fputc fputchar putc putcha ungetc	Записанный символ	EOF
данного типа int putw	Записанное целое значение	EOF
строки fputs puts	0	Ненулевое значение
форматизованных данных семейство функций printf	Число записанных симво- лов	Отрицательное значение

Функция `feof` определяет, достигнут ли конец потока. После того как достигнут конец потока, операции чтения из потока будут возвращать условие конца потока до тех пор, пока поток не будет закрыт или не будет выполнена функция `rewind` для потока. Использование функции позиционирования потока `fseek` не сбрасывает условие конца потока.

Функция `ferror` позволяет определить, произошла ли ошибка чтения-записи для потока. Если при выполнении операций чтения-записи для потока была обнаружена ошибка, то устанавливается индикатор ошибки для потока и он остается установленным до тех пор, пока поток не будет закрыт или индикатор ошибки не будет сброшен функцией `clearerr`.

Пример 4 представляет программу печати группы текстовых файлов. Программа иллюстрирует полный цикл работы с потоком, связанным с файлом на диске: открытие потока, чтение данных из потока, анализ возможных ошибочных ситуаций и закрытие потока при завершении работы с ним. Данные из текстового файла читаются построчно функцией `fgets`, которая возвращает значение `NULL` (определено в файле `stdio.h`), если произошла ошибка чтения или был обнаружен конец файла.

Пример 4.

```
/* Эта программа печатает группу файлов, имена которых
 * передаются в командной строке при вызове программы.
 * Каждый файл печатается с новой страницы, для него
 * выдается заголовок и ведется нумерация строк
 */
#include <stdio.h>      /* Требуется для объявления функций */
#include <stdlib.h>
#define ERRCODE    -1
#define NORMAL      0
#define MAXLINE    512
main(int argc, char ** argv)    /* Определение функции в
                                стандартном стиле */
{
    FILE * file_ptr;
    /* Если нет аргументов в командной строке, программа
       завершается с кодом -1 */
    if (argc == 1) {
        fprintf(stderr, "Нет аргументов\n");
        exit(ERRCODE);
    }
    else
        /* Цикл, пока не будут обработаны все аргументы командной
           строки */
```



```
while (--argc > 0) {
    /* Если файл открыт успешно, печатать его содержимое */
    if ((file_ptr = fopen(*++argv, "r")) != NULL) {
        char line[MAXLINE];
        int count;
        fprintf(stdprn, "\\f\\t\\tФайл %s\\n\\n\\n", *argv);
        for (count = 1; fgets(line, MAXLINE, file_ptr) !=
            NULL; count++)
            fprintf(stdprn, "%5d %s", count, line);
        if (!feof(file_ptr)) {
            fprintf(stderr, "Ошибка чтения файла %s\\n", *argv);
            fclose(file_ptr);
            continue;
        }
        fclose(file_ptr);
        printf("Файл %s напечатан\\n", *argv);
    }
    /* Если файл не может быть открыт, продолжить со следующего
       имени файла */
    else
        fprintf(stderr, "Файл %s не может быть открыт\\n", *argv);
}
return NORMAL;
}
```

11.2. УПРАВЛЕНИЕ БУФЕРИЗАЦИЕЙ

Потоки, открываемые функциями ввода-вывода потока, по умолчанию буферизованы, т.е. при открытии потока с ним связывается область памяти, называемая буфером. При операциях чтения блок данных из потока помещается во входной буфер и данные читаются из буфера. Когда входной буфер обработан, в буфер передается следующий блок данных. При операциях записи содержимое выходного буфера записывается в соответствующий поток (буфер выгружается) в следующих случаях:

- буфер заполняется;
- закрывается связанный с ним поток;
- при успешном завершении программы.

Буферизация повышает эффективность ввода-вывода, так как операционная система передает за одну операцию большие блоки данных вместо того, чтобы выполнять операцию ввода-вывода всякий раз, когда из потока читается или в него записывается элемент данных.

Буферизацией ввода-вывода управляют функции `setbuf` и `setvbuf`. Используя эти функции, можно сделать поток небуферизованным или связать буфер с небуферизованным потоком. Буферы, распределяемые операционной системой, пользователю недоступны, а буферами,

распределяемыми функциями `setbuf` и `setvbuf`, можно манипулировать как обычными переменными.

Функция `setbuf` связывает поток с буфером, который должен быть массивом символов длиной `BUFSIZ`. Константа `BUFSIZ` (определена в файле `stdio.h` и равна 512) задает размер буфера. Буфер, распределяемый функцией `setbuf`, используется для буферизации ввода-вывода вместо буфера, распределенного системой для данного потока. Чтобы сделать поток небуферизованным, вместо указателя буфера нужно задать нулевой указатель (`NULL`).

Ниже следует пример назначения собственного буфера потоку.

Пример.

```
#include <stdio.h>
FILE * fp;
char own_buf [BUFSIZ];
...
fp = fopen (...); /* Открытие потока */
...
setbuf (fp, own_buf); /* Назначить буфер own_buf потоку fp */
```

После выполнения этой последовательности операторов все операции ввода-вывода для потока `fp` будут осуществляться через буфер `own_buf`, а не через системный буфер.

Функция `setvbuf` позволяет назначать потоку буфер любого размера (но не превышающего 65535 байт). Она, кроме того, позволяет увеличить размер буфера, автоматически назначаемого потоку.

Примеры использования функции `setvbuf`:

```
FILE * fp;
char * buf;
unsigned int n;
setvbuf (fp, buf, _IOFBF, n); /* Назначение буфера
                               размером n потоку fp */
setvbuf (fp, NULL, _IOFBF, n); /* Увеличение размера
                               буфера, автоматически назначаемого потоку fp, на n */
setvbuf (fp, buf, _IONBF, n); /* Поток fp не буферизуется,
                               несмотря на buf и n */
```

Константы `_IOFBF` и `_IONBF`, определяющие тип операции над потоком, означают:

`_IOFBF` — поток буферизуется,

`_IONBF` — поток не буферизуется.

Стандартные потоки `stderr` и `stderr` не буферизованы, но, используя функции буферизации, их можно сделать буферизованными. Потоки `stdin`, `stdout` и `stderr` буферизованы; буфер выгружается, когда он заполнен или когда завершается функция, вызывающая ввод-вывод.

Следует иметь в виду, что если программа завершается аварийно, выходной буфер может быть не выгружен, в результате чего может происходить потеря данных.

Кроме описанных выше случаев, когда буфер автоматически выгружается, буфер может быть выгружен в любое время. Это позволяют сделать функции `fflush` и `flushall`. Функция `fflush` выгружает буфер отдельного указанного потока. Функция `flushall` выгружает содержимое всех буферов, связанных с открытыми выходными потоками. Если поток или потоки открыты для чтения, то функции `fflush` и `flushall` очищают содержимое буферов.

11.3. ДВОИЧНЫЙ И ТЕКСТОВЫЙ РЕЖИМЫ

Функции ввода-вывода потока позволяют открыть поток в текстовом или двоичном режиме. В текстовом режиме комбинация символов "возврат каретки — перевод строки" (символы `0x0d` — `0x0a`) преобразуются в одиночный символ перевода строки на вводе, а символ перевода строки преобразуется в комбинацию "возврат каретки — перевод строки" на выводе. Кроме того, комбинация клавиш УПР+Z (`0x1a`) интерпретируется как символ конца потока на вводе. В двоичном режиме преобразование символов "возврат каретки — перевод строки" подавляется. Текстовый режим задается литерой "t", а двоичный — литерой "b" в строковом литерале, определяющем тип доступа к потоку.

В примере 1 показано открытие потока, связанного с файлом на диске, для чтения и записи в двоичном режиме.

Пример 1.

```
#include <stdio.h>
```

```
...
```

```
FILE * fp;
```

```
...
```

```
fp = fopen ("data1", "rb+");
```

Порядок, в котором заданы литеры "b" и "+", не имеет значения. Например, строка "rb+" равносильна строке "r+b".

Если при открытии потока в строковом литерале не задана ни одна из литер "b" или "t", определяющая тип доступа к потоку, то по умолчанию поток открывается в текстовом режиме.

Стандартные потоки `stdin`, `stdout` и `stderr` открываются в текстовом режиме, а потоки `stdaux` и `stdprn` — в двоичном режиме.

Режим ввода-вывода для потока можно изменить после его открытия. Для этого предназначена функция ввода-вывода нижнего уровня `setmode`. Функции `setmode` передаются два аргумента: дескриптор файла и режим преобразования. Дескриптор файла представляет собой число, которое связывается с файлом и используется для ссылок на файл в последующих операциях с файлом. Функции потока используют указатель потока, а не дескриптор, поэтому сначала необходимо получить дескриптор файла, связанного с потоком, а затем использовать функцию `setmode`. Дескриптор файла, связанный с потоком, можно получить с помощью функции `fileno` библиотеки Си.

Пример 2 представляет функцию установки двоичного режима для потока, передаваемого как аргумент функции.

Пример 2.

```
#include <stdio.h>
#include <io.h>      /* Требуется только для использования */
#include <fcntl.h>    /* функции setmode */
#define ERROR      -1
#define NORMAL      0
switch_to_binary(FILE * file_ptr) /* Определение функции */
/* в стандартном стиле */
{
    if ((setmode(fileno(file_ptr), O_BINARY)) == ERROR) {
        fprintf(stderr, "Нельзя установить двоичный режим\n");
        return(ERROR);
    }
    else
        printf("Двоичный режим установлен\n");
    return(NORMAL);
}
```

Файлы `io.h` и `fcntl.h` требуются для использования функции `setmode`. Файл `io.h` содержит прототип функции `setmode`, а файл `fcntl.h` — определение констант, задающих режим переключения: `O_BINARY` (двоичный режим) и `O_TEXT` (текстовый режим).

Примечания.

1. Использование функций `ftell` и `fseek` для потока, открытого в текстовом режиме, может привести к неожиданным результатам из-за преобразования символов "возврат каретки — перевод строки". Значение указателя потока, возвращаемое функцией `ftell`, может не отражать физического смещения в байтах текущей позиции потока. Правильное выполнение функции `fseek` гарантируется только в следующих случаях:

- указатель потока устанавливается относительно базовых значений (`SEEK_SET`, `SEEK_CUR` и `SEEK_END`) со смещением 0;
- указатель потока устанавливается относительно начала потока (`SEEK_SET`) со значением смещения, возвращаемого функцией `ftell`.

2. Работа с данными двоичного вида в текстовом режиме иногда может привести к неожиданному условию конца потока (пример 3). Это может произойти в случае, если при чтении данных один из байтов оказывается символом `0x1a` (признак конца потока). Чтобы избежать подобных ситуаций, с данными двоичного вида следует работать в двоичном режиме.

Пример 3.

```
/* Эта простая программа демонстрирует неожиданный
 * результат, который можно получить при работе с файлом
 * в текстовом режиме
 */
#include <stdio.h>
char buf[] =
    "\x0\x1\x2\x3\x4\x5\x6\x7\x8\x9\xA\xB\xC\xD\xE\xF"
    "\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1A\x1B"
    "\x1C\x1D\x1E\x1F";
FILE * file_ptr;
main()
{
    int n;
    /* Открыть новый файл */
    if ((file_ptr = fopen("data.fil", "w+")) == NULL) {
        fprintf(stderr, "Ошибка открытия файла \"data.fil\"\n");
        return;
    }
    n = fwrite(buf, 1, 32, file_ptr);      /* Записать 32 байта в
                                           файл */
    printf("Записано %d байт(a) в файл \"data.fil\"\n", n);
    rewind(file_ptr);                     /* Установить на начало файла */
    n = fread(buf, 1, 32, file_ptr);      /* Прочитать 32 байта из
                                           файла */
    printf("Прочитано %d байт(a) из файла \"data.fil\"\n", n);
    fclose(file_ptr);
}
```

Вывод этой программы:

Записано 32 байт(a) в файл "data.fil"

Прочитано 26 байт(a) из файла "data.fil"

11.4. ФОРМАТИЗОВАННЫЙ ВВОД И ВЫВОД

Форматизованный ввод и вывод осуществляют два семейства функций: семейство функций `scanf` обеспечивает форматизованный ввод, а семейство функций `printf` — форматизованный вывод.

Семейство функций `scanf` включает:

`fscanf` — форматизованный ввод из потока,

`scanf` — форматизованный ввод из потока `stdin`,

`sscanf` — форматизованный ввод из строки Си.

Семейство функций `printf` состоит из следующих функций:

`fprintf`, `vfprintf` — форматизованный вывод в поток,

`printf`, `vprintf` — форматизованный вывод в поток `stdout`,

`sprintf`, `vsprintf` — форматизованный вывод в строку Си.

При форматизованном вводе-выводе данные, которые читаются или записываются, подвергаются определенным преобразованиям. Формат преобразований определяется строкой управления форматом, которая является строкой Си и задается как аргумент функций форматизованного ввода-вывода.

Форматизованный ввод

Функции форматизованного ввода предназначены для чтения данных из входного потока или строк Си и интерпретации их в соответствии со строкой управления форматом ввода. После преобразования данные записываются в области памяти, определенные аргументами, которые следуют за строкой управления форматом. Каждый аргумент должен быть указателем на переменную, тип которой соответствует типу, указанному в строке управления форматом.

Строка, управляющая форматом ввода, может состоять из следующих элементов:

- пробельных символов: пробела (' '), символа табуляции ('\t') или новой строки ('\n'). Пробельный символ указывает на то, что из входного потока надо читать, но не сохранять все последовательные пробельные символы вплоть до следующего непробельного символа. Один пробельный символ в строке управления форматом соответствует любому числу (включая нуль) или комбинации пробельных символов во входном потоке;

- непробельных символов, за исключением процента (%). Непробельный символ указывает на то, что надо читать, но не сохранять соответствующий ему непробельный символ. Если следующий символ во входном потоке не сравнился, функция форматизованного ввода завершается и конфликтный символ остается во входном потоке;

- спецификации формата с предшествующим символом процента (%). Спецификация формата указывает на то, что из входного потока надо читать символы и преобразовывать их в значение указанного типа. Значение присваивается аргументу в списке аргументов, следующем за строкой управления форматом.

Когда опознана первая спецификация формата, значение первого входного поля преобразуется в соответствии со спецификацией формата и сохраняется в области, указанной первым аргументом за строкой управления форматом. Следующая спецификация формата приводит к преобразованию следующего входного поля и сохранению в следующем аргументе и т.д. до конца строки, управляющей форматом ввода.

Входное поле составляют все выбираемые символы вплоть до первого пробельного символа или до первого символа, который не может быть преобразован в соответствии со спецификацией формата, или фиксированное количество символов, если оно указано, в зависимости от того, что произойдет раньше.

Если за строкой управления форматом аргументов больше, чем спецификаций формата, лишние аргументы игнорируются. Если для спецификаций формата недостаточно аргументов, результат не определен.

Спецификация формата имеет следующий вид:

`%[*][длина][F|N][h|l]тип`

Каждое поле спецификации формата является одиночным символом или числом, означающим конкретный режим формата. Символ *типа* определяет, как интерпретируется входное поле.

Если за символом процента (%) следует символ, который не является символом управления форматом, этот символ и все следующие символы (до следующего символа процента) рассматриваются как обычная последовательность символов, т.е. последовательность символов, которая должна совпадать со вводом. Например, чтобы указать символ процента, надо использовать %%.

Звездочка (*), следующая за символом процента, подавляет присваивание следующего входного поля, которое интерпретируется как поле указанного типа. Поле читается, но не сохраняется.

Поле *длина* (положительное десятичное целое число) задает максимальное число символов, которое может быть прочитано из входного потока. Из входного потока читается заданное количество символов, если только раньше не встретится пробельный символ или символ, который не может быть преобразован в соответствии с заданным форматом. В последнем случае может быть прочитано меньше символов, чем требуется в спецификации длины.

Префиксы F и N позволяют подавить соглашения по умолчанию об адресации используемой модели памяти. Префикс F должен использоваться для аргумента, указывающего на объект *far*, а префикс N — для аргумента, указывающего на объект *near*. Префикс l означает, что соответствующий спецификации формата аргумент должен указывать на объект типа *long* или *double*, а префикс h означает, что аргумент должен быть указателем на тип *short*.

Символы типа, которые могут быть заданы в спецификации формата, приведены в табл. 11.3.

Таблица 11.3

Символы типа в спецификации формата ввода

Символ типа	Ожидаемый тип ввода	Тип аргумента
d	Десятичное целое	Указатель на int
o	Восьмеричное целое	То же
x	Шестнадцатеричное целое	-"
i	Десятичное, восьмеричное или шестнадцатеричное целое	-"
u	Десятичное целое без знака	Указатель на unsigned int
U	Десятичное целое длинное без знака	Указатель на unsigned long
e,E,f,g,G	Значение с плавающей точкой вида: [+ -] dddd [E e [+ -] dd] где dddd — одна или более десятичных цифр, возможно содержащих десятичную точку, dd — целое	Указатель на float
c	Символ. Если во входном потоке встретится пробельный символ, он будет читаться, а не пропускаться. Для чтения следующего непробельного символа следует использовать спецификацию %ls	Указатель на char
s	Строка символов. Для чтения строк, не ограниченных пробельными символами, вместо символа типа s следует использовать набор символов в квадратных скобках. Символы из входного потока читаются до первого символа, отличного от символов в квадратных скобках. Если же первым символом в квадратных скобках задан символ ^, то символы из входного потока читаются до первого символа из квадратных скобок	Указатель на массив символов, размер которого достаточен для сохранения строки и нулевого символа конца строки (\0), который присоединяется автоматически
p	Никаких данных не читается	Указатель на объект типа int, в который будет помещено количество успешно прочитанных символов (до данной спецификации)
r	Значение вида xxxx:yyyy (сегмент:смещение), где x,y — прописные шестнадцатеричные цифры	Указатель на указатель far на void

Пример.

```
#include <stdio.h>
char str[] = "Из этой строки читаются:
    десятичное целое -125,
    шестнадцатеричное целое 25c,
    целое без знака 65535,
    значение с плавающей точкой -1233e-2,
    несколько строк и символов и указатель far 25cc:af50";
```



```

int i, x, n1, n2;
unsigned int ui;
float f;
char s1[30], s2[30];
char c1, c2;
void far * p;
n1 = sscanf(str, "Из этой строки читаются:"
" %*10s %*5s %d, %*17s %*5s %x, %*s %*s %*s %u, "
"значение с плавающей точкой %e, %*s %s %*1s %[^ол] %*4s%c "
"%*s%*s%1sar %p%n", &i, &x, &ui, &f, s1, s2, &c1, &c2, &p, &n2);

```

В результате выполнения функции `sscanf` переменные будут иметь следующие значения:

i — -125,
x — 25с,
ui — 65535,
f — -12.33,
s1 — строка,
s2 — симв,
c1 — (пробел),
c2 — f,
p — 25сс:af50,
n1 — 9 (количество успешно преобразованных и присвоенных входных полей),
n2 — 209 (количество успешно прочитанных символов).

Форматизованный вывод

Функции форматизованного вывода преобразуют данные в соответствии со строкой управления форматом вывода и выводят их в выходной поток или в строку Си. Данные, которые преобразуются и выводятся, задаются как аргументы функций форматизованного вывода после строки управления форматом.

Строка, управляющая форматом вывода, может содержать обычные символы, управляющие последовательности и спецификации формата. Обычные символы и управляющие последовательности просто копируются в выходной поток (строку Си).

Спецификация формата:

`%[флажок][длина][.точность][F|N|h|I]тип`

Каждое поле спецификации формата является одиночным символом или числом, задающим конкретный режим формата. Если за знаком % следует символ, который не является полем формата, то символ просто копируется в выходной поток.

Символы типа, которые могут быть использованы в спецификации формата вывода, приведены в табл. 11.4.

Таблица 11.4

Символы типа в спецификации формата вывода

Символ типа	Тип аргумента	Формат вывода
d	int	Десятичное целое со знаком
i	int	Десятичное целое со знаком
u	int	Десятичное целое без знака
o	int	Восьмеричное целое без знака
x	int	Шестнадцатеричное целое без знака, использующее "abcdef"
X	int	Шестнадцатеричное целое без знака, использующее "ABCDEF"
f	double	Значение со знаком в формате [-]dddd.dddd, где dddd — одна или более десятичных цифр. Количество цифр перед десятичной точкой зависит от величины числа, а количество цифр после десятичной точки зависит от требуемой точности
e	double	Значение со знаком в формате [-]d.dddde[+ -]ddd, где d — одна десятичная цифра, dddd — одна или более десятичных цифр
E	double	Идентичен формату "e", за исключением того, что экспоненту представляет "E"
g	double	Значение со знаком печатается в формате "f" или "e" в зависимости от того, какой из них более компактен для данного значения и точности. Формат "e" используется только, когда значение экспоненты меньше минус 4 или больше заданной точности. Конечные нули усекаются, а десятичная точка появляется, если за ней следует хотя бы одна цифра
G	double	Идентичен формату "g", за исключением того, что экспоненту представляет "G"
c	int	Одиночный символ
s	Строка	Символы печатаются до первого нулевого символа ('\0') или до достижения значения заданной точности
p	Указатель на целое	Количество символов, успешно записанных в поток или в строку. Это значение не выводится
r	Указатель far на void	Печатает адрес, указанный аргументом, в формате xxxx:yyyy, где xxxx — сегмент, yyyy — смещение, а "x" и "y" — строчные шестнадцатеричные цифры

Символ *флажка* управляет выравниванием вывода и печатью знака числа, пробелов, десятичной точки, префиксов восьмеричной и шестнадцатеричной систем счисления. В спецификации формата может

быть задано более одного флажка. Символы флажка в спецификации формата приведены в табл. 11.5.

Таблица 11.5

Символы флажка в спецификации формата вывода

Символ флажка	Значение	Значение по умолчанию
-	Выравнивание влево внутри заданного поля	Выравнивание вправо
+	Присоединение знака (+ или -) к выводимому значению, если оно имеет тип со знаком	Знак (-) появляется только для отрицательных значений со знаком
Пробел	Присоединение пробела к выводимому значению, если оно имеет тип со знаком и положительно; пробел игнорируется, если одновременно указаны флажки "пробел" и "+"	Пробела нет
#	Если используется с форматами "о", "х" или "Х", любое ненулевое значение выводится с предшествующим 0, 0х или 0Х соответственно Если используется с форматами "е", "Е" или "f", выводимое значение во всех случаях будет содержать десятичную точку Если используется с форматами "g" или "G", выводимое значение во всех случаях будет содержать десятичную точку и подавляется усечение конечных нулей Игнорируется, когда используется с форматом "с", "d", "i", "u" или "s"	Пробела нет Десятичная точка появляется, если только за ней следуют цифры Десятичная точка появляется, если только за ней следуют цифры, конечные нули усекаются

Поле *длина*, задаваемое положительным целым числом, определяет минимальное число выводимых символов. Если количество символов в выводимом значении меньше, чем задано в поле длины, выводимое значение дополняется пробелами до заданной минимальной длины. Если поле длины задано с начальным нулем, нули добавляются к выводимому значению до заданного минимального размера.

Спецификация минимального размера никогда не приводит к усечению выводимого значения: если количество символов в выводимом значении больше указанного размера или минимальный размер не задан, выводятся все символы значения (в соответствии с полем "точность").

Поле *точность*, задаваемое положительным целым числом с предшествующей точкой, определяет количество выводимых символов,

количество десятичных позиций или количество значащих цифр. В отличие от поля длины, поле точности может привести к усечению или округлению выводимого значения.

Интерпретация значений поля точности в спецификации формата описана в табл. 11.6, а назначение префиксов F, N, h, l в спецификации формата вывода — в табл. 11.7.

Таблица 11.6

Интерпретация значений поля точности

Тип	Значение точности	Значение по умолчанию
d i u o x X	Точность указывает минимальное количество выводимых цифр. Если количество цифр в аргументе меньше, чем точность, выводимое значение слева дополняется нулями. Если количество цифр превышает точность, значение не усекается	Если точность равна нулю или опущена или за точкой нет цифр, точность устанавливается равной 1
e E	Точность указывает количество выводимых цифр после десятичной точки. Последняя выводимая цифра округляется	По умолчанию точность равна 6; если точность равна нулю или за точкой нет цифр, десятичная точка не выводится
f	Точность указывает количество цифр после десятичной точки. Если десятичная точка появляется, ей предшествует по меньшей мере одна цифра. Значение округляется в соответствии с количеством цифр	По умолчанию точность равна 6; если точность равна нулю или за точкой нет цифр, десятичная точка не выводится
g G	Точность указывает максимальное количество значащих цифр, которые будут выводиться	Выводятся 6 значащих цифр
c	Не имеет значения	Выводится символ
s	Точность указывает максимальное количество выводимых символов. Символы, превышающие точность, не печатаются	Выводятся все символы до нулевого символа

Следующий пример демонстрирует использование функции printf для назначения функциональным клавишам некоторых команд операционной системы и драйверов.

Таблица 11.7

Назначение префиксов F, N, h, l в спецификации формата вывода

Символ префикса	Назначение
F, N	Префикс, который дает возможность пользователю подавить соглашения по умолчанию об адресации в используемой модели памяти; F используется для печати значений, которые были объявлены как far в малой модели памяти; N предназначен для значений near в средней, большой и сверхбольшой моделях памяти F и N должны использоваться только с символами типа "s" и "p", поскольку они относятся только к аргументам, которые передают указатели
h, l	Префиксы, которые определяют размер ожидаемого аргумента следующим образом: h используется как префикс с типами d, i, o, x и X — для указания того, что тип аргумента short int, или с "u" — для указания того, что тип аргумента short unsigned int; l используется как префикс с типами d, i, o, x и X — для указания того, что тип аргумента long int, или с типом "u" — для указания того, что тип аргумента long unsigned int; он также используется с типами e, E, f, g и G — для указания того, что тип аргумента double, а не float

Пример.

```
/* Эта программа назначает функциональным клавишам
* следующие команды:
*   Ф2 — A: + ВВОД
*   Ф4 — TYPE
*   Ф5 — COPY
*   Ф6 — DEBUG + ВВОД
*   Ф7 — MOUSE + ВВОД
*   Ф8 — CLS + ВВОД
*   Ф9 — EXIT + ВВОД
*   Ф10 — DIR + ВВОД
* Примечание.
*   Переопределение функциональных клавиш осуществляется
*   путем посылки в драйвер ANSI.SYS управляющих последовательностей вида:
*   ESC [<расширенный код клавиши>;<определение клавиши>p
*   До вызова этой программы должен быть загружен драйвер ANSI.SYS,
*   т. е. файл конфигурации CONFIG.SYS
*   должен содержать следующую запись:
*   DEVICE=ANSI.SYS
*   Драйвер ANSI.SYS описан в книге В. М. Брябрина
*   "Программное обеспечение персональных ЭВМ" [8]
*/
```



```
#include <stdio.h>      /* Требуется для функции printf */
main()
{
    printf("\x01b[0;60;\xA: ";13p");
    printf("\x01b[0;62;\TYPE ";p");
    printf("\x01b[0;63;\COPY ";p");
    printf("\x01b[0;64;\DEBUG ";13p");
    printf("\x01b[0;65;\MOUSE ";13p");
    printf("\x01b[0;66;\CLS ";13p");
    printf("\x01b[0;67;\EXIT ";13p");
    printf("\x01b[0;68;\DIR ";13p");
}
```

11.5. ВВОД-ВЫВОД НИЖНЕГО УРОВНЯ

Функции ввода-вывода нижнего уровня, в отличие от функций ввода-вывода потока, выполняют небуферизованный и неформатизованный ввод-вывод и прямо вызывают средства ввода-вывода операционной системы. С файлом, открываемым функцией нижнего уровня, связывается дескриптор файла, используемый операционной системой для ссылки на файл.

Список и прототипы функций ввода-вывода нижнего уровня приведены в приложении 2.

Для применения функций нижнего уровня необходимо включение файла `io.h`.

Открытие и закрытие файлов

Перед выполнением операций ввода-вывода файл должен быть открыт одной из следующих функций:

`creat` — создает новый файл;

`open` — открывает файл;

`open` — открывает файл для совместного использования.

Файл может быть открыт для чтения, записи или чтения и записи в двоичном или текстовом режиме. Функция открытия возвращает дескриптор файла, который можно присвоить целой переменной и затем использовать эту переменную для ссылок на файл.

Пример показывает открытие файла функцией нижнего уровня.

Пример.

```
#include <stdio.h>
#include <io.h>      /* Требуется для объявлений функций
                      нижнего уровня */
#include <fcntl.h>    /* Требуется для определения констант,
                      используемых в функциях открытия */

...
int handle;
...
if ((handle = open("data", O_RDONLY)) == -1)
    fprintf(stderr, "Ошибка открытия файла \n");
```


Второй аргумент функции `open` задает тип операций, допустимых для открываемого файла, и является выражением, образованным комбинацией одной или более следующих предопределенных констант:

- `O_APPEND` — файл открывается для записи в конец файла;
- `O_BINARY` — файл открывается в двоичном режиме;
- `O_CREAT` — создается и открывается новый файл;
- `O_RDONLY` — файл открывается только для чтения;
- `O_RDWR` — файл открывается и для чтения, и для записи;
- `O_TEXT` — файл открывается в текстовом режиме;
- `O_TRUNC` — открывается существующий файл и его содержимое удаляется;
- `O_WRONLY` — файл открывается только для записи.

Когда задано более одной константы, то они связываются поразрядной операцией ИЛИ (`|`).

Примечание. Так как режим доступа по умолчанию не устанавливается, при открытии файла должна быть обязательно задана одна из констант: `O_RDONLY`, `O_RDWR` или `O_WRONLY`. Так, если надо открыть новый файл для записи, вызов функции `open` из предыдущего примера должен иметь, например, такой вид:

```
handle = open("data", O_CREAT | O_WRONLY);
```

С началом выполнения Си-программы автоматически открываются пять файлов, соответствующих стандартному вводу, стандартному выводу, стандартному выводу сообщений об ошибках, стандартному дополнительному выводу и стандартной печати, и для них определяются следующие предопределенные дескрипторы:

- 0 — для `stdin`,
- 1 — для `stdout`,
- 2 — для `stderr`,
- 3 — для `stdaux`,
- 4 — для `stdprn`.

Применяя эти предопределенные в системе дескрипторы, программа может иметь доступ к стандартным файлам.

Одному и тому же файлу можно назначать несколько дескрипторов. Это позволяют сделать функции `dup` и `dup2`:

- `dup` — создает новый дескриптор для файла;
- `dup2` — заменяет дескриптор файла.

Используя функцию `dup2`, можно связать предопределенный дескриптор, например, с файлом на диске. Следующий пример показывает назначение дополнительного дескриптора потоку.

Пример.

```
#include <io.h>      /* Требуется для функций нижнего уровня */
#include <stdio.h>    /* Требуется только для функции fileno */
int cause_handle (FILE * file_ptr)
{
    int new;
```



```

if ((new = dup (fileno(file_ptr))) == -1)
    printf(stderr, "Нельзя назначить дополнительный
                дескриптор потока\n");
return(new);
}

```

Чтение и запись данных

Для чтения и записи данных предназначены две функции: `read` и `write`. Как и для ввода-вывода потока, операции чтения и записи нижнего уровня всегда начинаются с текущей позиции файла. Каждый раз, когда выполняется чтение или запись, текущая позиция файла изменяется.

Для анализа условия конца файла может использоваться функция `eof`. Функция `lseek` позволяет позиционировать файл. Следующая операция ввода-вывода будет выполняться с указанной позиции. Для определения текущей позиции файла используется функция `tell`. Функции `lseek` и `tell` нельзя применять к файлам, представляющим устройства.

В следующем примере показан прямой ввод из файла.

Пример.

```

/* Функция read_file читает nbytes байт с позиции pos из
 * файла, имя которого передается функции как аргумент.
 * Функция сохраняет прочитанные байты в массиве save
 * и возвращает число прочитанных байтов
 */
#include <io.h>      /* Требуется для функций нижнего уровня */
#include <stdio.h>    /* Требуется для функций lseek и fprintf */
#include <fcntl.h>    /* Требуется для функции open */
#define ERROR -1
long read_file(filename, nbytes, pos, save)
char * filename;      /* Имя файла */
long nbytes;          /* Число читаемых байтов */
long pos;             /* Начальная позиция */
char * save;          /* Буфер для чтения */
{
    int fh;
    int n;
    /* Открыть файл */
    if ((fh = open(filename, O_RDONLY | O_BINARY)) == ERROR) {
        fprintf(stderr, "Ошибка открытия файла \"%s\"\n",
            filename);
        return(ERROR);
    }
    /* Установить позицию для чтения */
    if (lseek(fh, pos, SEEK_SET) == ERROR) {
        fprintf(stderr, "Ошибка позиционирования файла \"%s\"\n",
            filename);
        return(ERROR);
    }
}

```



```
/* Прочитать nbytes байт из файла */  
n = read(fh, save, nbytes);  
if (eof(fh))  
    fprintf(stderr, "Достигнут конец файла \"%s\"\\n",  
            filename);  
/* Заккрыть файл */  
close(fh);  
/* Вернуть число прочитанных байтов */  
return(n);  
}
```

11.6. ВВОД И ВЫВОД ДЛЯ КОНСОЛИ И ПОРТА

Функции ввода-вывода для консоли (т.е. дисплей плюс клавиатура) и порта могут рассматриваться как расширение функций ввода-вывода потока. Они позволяют выполнить операции чтения и записи для консоли и порта ввода-вывода.

Под операцией ввода-вывода для консоли понимается либо ввод данных с клавиатуры, либо вывод данных в текущую строку экрана дисплея. Функции ввода-вывода для консоли позволяют прочитать и записать отдельные символы, строки символов и форматизованные данные.

Для ввода-вывода с консоли предназначены функции:

- cgets — вводит строку с консоли;
- sprintf — выводит форматизованные данные на консоль;
- cputs — выводит строку на консоль;
- cscanf — вводит форматизованные данные с консоли;
- getch — вводит символ с консоли;
- getche — вводит символ с консоли и отображает его;
- kbhit — проверяет, была ли нажата клавиша на консоли;
- putch — выводит символ на консоль;
- ungetch — "возвращает" прочитанный символ на консоль.

Функции ввода-вывода для порта просто читают или пишут один или два байта в указанный порт.

Для ввода-вывода в порт предназначены функции:

- inr — читает байт из порта;
- inrw — читает слово (2 байта) из порта;
- outp — записывает байт в порт;
- outpw — записывает слово (2 байта) в порт.

Перед выполнением операций ввода-вывода консоль и порт не открываются и после операций ввода-вывода не закрываются, поэтому для консоли и портов нет функций открытия и закрытия.

Использование функций ввода-вывода для консоли и порта требует включения файла `conio.h`.

Пример 1 представляет программу чтения с консоли и отображения кода прочитанной клавиши.

Пример 1.

```
#include <conio.h>      /* Требуется для объявления функций */
                        /* ввода-вывода с консоли */
#include <stdio.h>      /* Требуется для функции printf */
main()
{
    char c;
    /* Цикл, пока не будет прочитана комбинация клавиш УПР+Z */
    while ((c = getch()) != 0x1a) {
        if (c == 0 || c == 0xe0)
            printf("Код специальной клавиши или комбинации"
                "клавиш — %1x\n", getch());
        else
            printf("Код прочитанного символа — %1x\n", c);
    }
}
```

Для чтения специальной клавиши, такой как функциональная клавиша или клавиша перемещения курсора, функция `getch` (`getche`) должна быть вызвана дважды.

В примере 2 приведен фрагмент функции, реализующей обработчик прерываний от клавиатуры. Обработчик должен выполнять следующую последовательность действий:

- 1) сохранить содержимое регистров процессора в стеке;
- 2) прочитать код клавиши из вводного порта 0x60 трехканального порта. Через порт 0x60 передаются сигналы, принимаемые от клавиатуры;
- 3) сбросить прерывание от клавиатуры. Прерывание от клавиатуры сбрасывается путем установки в единичное состояние разряда 7 выводного порта 0x61 трехканального порта. После этого клавиатура может посылать следующий символ процессору;
- 4) обработать код клавиши. Следует иметь в виду, что клавиатура посылает процессору не код клавиши в наборе кодов ПЭВМ, а скэн-код клавиши. Каждая клавиша имеет свой собственный скэн-код. Например, клавише Вых соответствует скэн-код 1, клавише "1" — 2 и т.д. Таким образом, порт содержит скэн-код и его нужно перекодировать в правильный код символа. Когда на клавиатуре нажимается клавиша, процессору посылается не один, а два сигнала: при нажатии клавиши посылается код нажатия клавиши (скэн-код), а когда клавиша отпускается — код отпускания клавиши. Он создается путем прибавления числа 128 к коду нажатия клавиши. Клавиша считается нажатой до тех пор, пока не будет получен код отпускания;
- 5) когда обработка кода клавиши завершена, необходимо послать команду EOI (конец прерывания) в контроллер прерываний. Эта ко-

манда состоит в следующем: послать значение 0x20 в порт 0x20 контроллера прерываний;

б) восстановить содержимое регистров процессора и выполнить команду IRET.

Команды сохранения и восстановления регистров процессора, а также команда IRET будут генерироваться компилятором автоматически благодаря тому, что в объявлении функции указан атрибут `interrupt`.

Пример 2.

```
/* В этой программе показано, как можно оформить функцию Си
 * в виде обработчика прерываний и как оставить ее резидентной
 * в памяти. Кроме того, программа демонстрирует использование:
 * 1. Функций ввода-вывода для порта
 *      inp
 *      outp (прототип в conio.h)
 * 2. Функций интерфейса с Альфа-ДОС
 *      _dos_setvect
 *      _dos_keep
 *      _dos_freemem (прототипы в dos.h)
 * 3. Встроенных в Си-программу команд на ассемблере
 * 4. Прагм и глобальных переменных Си
 * а также отключение некоторых стандартных функций
 * инициализации
 */
```

```
#include <dos.h>
#include <conio.h>
#define KEYBRD_INT 0x9
/* Отключить проверку стека и указателей */
#pragma check_stack(off)
#pragma check_pointer(off)
/* Прототип функции обработчика прерываний */
void far interrupt keybrd_handler(void);
/* Ссылки на глобальные переменные */
extern unsigned int near _psp;
/* Содержит сегментный адрес префикса программного сегмента
   (текущей программы) */
extern unsigned int near _amblksiz;
/* Управляет размером области памяти для динамического
   распределения памяти (кучи) */
/* Указатели на вершину стека и PSP программы */
char huge *tsr_top; /* Объявлены как указатели huge для */
char huge *tsr_bottom; /* того, чтобы из одного указателя */
/* можно было вычесть другой */
/* Подавить обработку аргументов командной строки и
 * переменных операционной среды, что позволит несколько
 * сократить требования программы к объему памяти
 */
_setargv()
{
}
```



```

_setenvp()
{
}
main()
{
    unsigned int tsr_size; /* Размер программы */
    _asm mov WORD PTR tsr_top[0], ss /* Установить указатель */
    _asm mov WORD PTR tsr_top[2], sp /* tsr_top на вершину */
                                   /* стека */
    FP_SEG(tsr_bottom) = _psp; /* Установить указатель */
    FP_OFF(tsr_bottom) = 0x0; /* tsr_bottom на PSP */
                                   /* программы */
    _amblksiz = 256; /* 16 параграфов для кучи */
    /* Размер программы (в параграфах) равен:
     * преобразованной в параграфы разности (адрес вершины стека
     * — адрес PSP программы) + число параграфов в куче +
     * один дополнительный параграф для надежности
     */
    tsr_size = ((tsr_top - tsr_bottom) >> 4) + (_amblksiz >> 4) + 1;
    /* Связать вектор 0x9 с собственным обработчиком прерываний
     * от клавиатуры
     */
    dos_setvect(KEYBRD_INT, keybrd_handler);
    /* Завершить программу и оставить ее резидентной */
    _dos_keep(0, tsr_size);
}
#define INT_CONTROLLER 0x20
#define PPI_PORT_A 0x60
#define PPI_PORT_B 0x61
void interrupt far keybrd_handler(void)
{
    unsigned char scan_code;
    unsigned char byte1, byte2;
    /* Прочитать скэн-код клавиши */
    scan_code = inp(PPI_PORT_A); /* Прочитать байт из порта */
                                   /* 0x60 */
    /* Сбросить прерывание от клавиатуры */
    byte1 = byte2 = inp(PPI_PORT_B);
                                   /* Прочитать и сохранить байт из порта 0x61 */
    byte1 |= 0x80; /* Установить в 1 разряд 7 порта 0x61 */
    outp(PPI_PORT_B, byte1);
    outp(PPI_PORT_B, byte2); /* Восстановить исходное значение */
    . Текст обработчика прерывания
    /* Послать команду EOI (конец прерывания) в контроллер
     прерываний */
    outp(INT_CONTROLLER, 0x20);
}

```


11.7. ДРУГИЕ ФУНКЦИИ ВВОДА-ВЫВОДА

Кроме функций ввода-вывода потока и нижнего уровня (функции ввода-вывода для консоли и порта можно рассматривать как расширение функций потока), библиотека Си предоставляет еще одну категорию функций ввода-вывода — функции, которые для выполнения операций ввода-вывода используют системные вызовы операционной системы Альфа-ДОС:

- `_dos_close` — закрывает файл (системный вызов `0x3E`);
- `_dos_creat` — создает новый файл (системный вызов `0x3C`);
- `_dos_creatnew` — создает новый файл (системный вызов `0x5B`);
- `_dos_findfirst` — находит первый файл по заданному шаблону имени файла (системный вызов `0x4E`);
- `_dos_findnext` — находит следующий файл по заданному шаблону имени файла (системный вызов `0x4F`);
- `_dos_getfileattr` — получает атрибуты файла (системный вызов `0x43`);
- `_dos_open` — открывает файл (системный вызов `0x3D`);
- `_dos_read` — читает файл (системный вызов `0x3F`);
- `_dos_setfileattr` — устанавливает атрибуты файла (системный вызов `0x43`);
- `_dos_write` — пишет в файл (системный вызов `0x40`).

Функции интерфейса с Альфа-ДОС требуют включения файла `dos.h`.

Эти функции используются в следующем примере программы сохранения таблицы разделов жесткого диска. Первый сектор диска содержит основной блок начальной загрузки. В конце этого сектора расположена таблица разделов диска. Эта таблица состоит из четырех записей по 16 байт каждая (т.е. диск можно разбить максимум на 4 области). Если диск разбит на разделы, то запись таблицы разделов содержит информацию о соответствующем разделе диска, как показано в табл. 11.8.

Во время загрузки операционной системы BIOS считывает с диска и выполняет основной блок начальной загрузки, если она установит, что на дисковомодеме А нет дискеты. Основной блок начальной загрузки считывает таблицу разделов диска и определяет активный раздел диска. Затем из активного раздела считывается блок начальной загрузки и ему передается управление.

Так как первый сектор диска содержит информацию о распределении диска, то желательно иметь копию этой информации. Программа, текст которой приведен ниже, позволяет сохранить копию первого сектора диска в файле на дискете. Копия сохраняется в файле с именем `MASTER.FIL`, который защищен от случайного удаления. Кроме

того, эта программа демонстрирует функции ввода-вывода, использующие системные вызовы Альфа-ДОС, и функции интерфейса с BIOS.

Таблица 11.8

Структура записи таблицы разделов диска

Смещение (шестнадцатеричное)	Длина поля	Содержимое
0	1	Флажок блока начальной загрузки: 0 — не активный 80H — активный
1	1	Номер начальной головки
2	2	Номера сектора и цилиндра блока начальной загрузки
4	1	Длина элемента таблицы размещения файлов: 1 — 12-битовый элемент 4 — 16-битовый элемент
5	1	Номер конечной головки
6	2	Номера сектора и цилиндра последнего сектора раздела
8	4	Относительный номер первого сектора
c	4	Размер раздела в секторах

Пример.

```
#include <stdio.h> /* Требуется для объявления функций */
/* потока, */
#include <ctype.h> /* toupper, */
#include <conio.h> /* ввода-вывода с консоли, */
#include <bios.h> /* интерфейса с BIOS, */
#include <dos.h> /* интерфейса с Альфа-ДОС, */
#include <fcntl.h> /* для определения констант */
#define ERRCODE -1
#define NORMAL 0
/* Структуры, используемые функциями */
/* интерфейса с BIOS и Альфа-ДОС: */
struct diskinfo_t disk_info; /* содержит параметры диска */
/* (объявлена в bios.h), */
* struct diskfree_t disk_space; /* информацию о свободной */
/* памяти на диске */
/* (объявлена в dos.h) */
char sector_buf[512]; /* Буфер для чтения сектора */
/* диска */
char file_name[] = "x:MASTER.FIL"; /* Имя файла, в котором */
```



```

/* будет сохраняться */
/* копия первого */
/* сектора диска */

main()
{
char drive_letter; /* Имя устройства */
int file_handle; /* Дескриптор файла */
int status; /* Рабочие переменные */
unsigned bytes;
unsigned long far * fp = NULL;
/* Сначала необходимо убедиться в том, что жесткий диск */
/* присутствует в конфигурации машины. Для этого */
/* используется вектор прерываний для INT 41H таблицы */
/* векторов прерываний. Если диск присутствует в */
/* конфигурации машины, то в этом векторе хранится */
/* адрес таблицы параметров диска. В противном случае */
/* вектор для INT 41H будет содержать нуль. */
*((unsigned *) &fp) = 0x104; /* Установить fp на адрес */
/* вектора INT 41H */

*((unsigned *) &fp + 1) = 0x0;
if (*fp == NULL) /* Завершить программу, если диска нет */
    fprintf(stderr, "Нет диска в конфигурации\n");
    return(ERRCODE);
}
/* Определить имя устройства, на котором будет сохранена
таблица разделов диска */
drive_letter = ' ';
printf("Введите имя устройства для копии: ");
drive_letter = getch();
drive_letter = toupper(drive_letter);
while (drive_letter < 'A' || drive_letter > 'D') {
    printf("\nВведите имя устройства (A, B, C или D): ");
    drive_letter = getch();
    drive_letter = toupper(drive_letter);
}
putchar('\n');
printf("Поставьте дискету на устройство %c и нажмите любую
клавишу\n", drive_letter);
getch();
/* Проверить, есть ли место на дискете для сохранения
* таблицы разделов. Получить количество свободных
* кластеров на дискете */
while (1) {
    _dos_getdiskfree(drive_letter - 'A' + 1, &disk_space);
    if (disk_space.avail_clusters > 0)
        break;
    printf("Нет места на дискете\n");
    printf("Поставьте новую дискету на устройство %c и нажмите
любую клавишу\n", drive_letter);
    getch();
}
/* Прочитать таблицу разделов диска во временный буфер */
/* Установить параметры для чтения первого сектора диска:*/

```



```

disk_info.drive = 0x80; /* жесткий диск */
disk_info.head = 0x0; /* номер головки */
disk_info.track = 0x0; /* номер дорожки */
disk_info.sector = 0x1; /* номер начального сектора */
disk_info.nsectors = 0x1; /* число читаемых секторов */
disk_info.buffer = sector_buf; /* буфер для чтения */
status = _bios_disk(_DISK_READ, disk_info); /* Читать
                                             сектор */
if (status) { /* Завершить программу, если ошибка чтения */
    printf("Ошибка чтения с диска\n");
    return(ERRCODE);
}
/* Сохранить прочитанный сектор в файле MASTER.FIL
   на дискете */
*file_name = drive_letter;
/* Создать файл MASTER.FIL */
status = _dos_creat(file_name, _A_NORMAL, &file_handle);
if (status) { /* Завершить программу, если нельзя */
              /* создать файл MASTER.FIL */
    printf("Нельзя создать файл \"%s\"\n", file_name);
    return(ERRCODE);
}
/* Открыть файл MASTER.FIL */
_dos_open(file_name, O_WRONLY, &file_handle);
/* Записать сектор в файл MASTER.FIL */
_dos_write(file_handle, disk_info.buffer, BUFSIZ, &bytes);
_dos_close(file_handle); /* Закрыть файл MASTER.FIL и */
                        /* установить для него атрибут */
                        /* "только для чтения" */
_dos_setfileattr(file_name, _A_RDONLY);
/* Завершить программу */
printf("Таблица разделов диска сохранена в файле "
       "\"%s\"", file_name);
return(NORMAL);
}

```

11.8. ОГРАНИЧЕНИЯ ПО ИСПОЛЬЗОВАНИЮ ФУНКЦИЙ ВВОДА-ВЫВОДА

Функции потока и функции нижнего уровня в общем случае несовместимы, поэтому при работе с файлом должны использоваться либо функции потока, либо функции нижнего уровня. В связи с тем что функции потока выполняют буферизацию, а функции нижнего уровня нет, доступ к одному и тому же файлу двумя разными способами может привести к потере данных в буфере.

Функции ввода-вывода с консоли несовместимы с функциями потока и нижнего уровня и не должны использоваться вместе с ними.

Функции ввода-вывода, использующие интерфейс с Альфа-ДОС, не должны использоваться с функциями ввода-вывода потока, нижнего уровня или с консоли.

Глава 12

РАБОТА С ОКНАМИ

Библиотека функций Си предоставляет набор функций, которые обеспечивают возможность разрабатывать Си-программы, предназначенные для создания и управления окнами на экране дисплея при работе в текстовом режиме. Функции работы с окнами позволяют выполнять такие операции, как создание окон, установка их на экран и удаление с экрана, вывод в окно и чтение из окна, перемещение окон по экрану, сохранение образа окна в памяти и восстановление его на экране, просмотр текстовых файлов в окне, организация меню, файлов справочной информации, буферов виртуального экрана и др. При этом обеспечивается высокая скорость выполнения этих операций.

Ниже дается краткий обзор средств и возможностей, предоставляемых системой поддержки окон. Полный список функций для работы с окнами и их прототипы можно найти в приложении 2.

12.1. ОПРЕДЕЛЕНИЕ ОКНА

Определение окна в Си-программе включает в себя объявление окна и определение характеристик (или параметров) окна.

Объявление окна связывает окно со структурой предопределенного типа WINDOW, которая определяет окно, и имеет вид:

```
WINDOW имя-окна;
```

Например, спецификация:

```
WINDOW wn1, wn2;
```

объявляет два окна — wn1 и wn2.

Структура типа WINDOW, связываемая с окном, будет содержать в виде элементов структуры все параметры данного окна. Параметры окна определяют координаты окна на экране, его размер, вид рамки, цвет и другие свойства окна, поэтому следующим шагом после объявления окна является определение параметров окна.

Для определения параметров окна используются две функции: def_wn и defs_wn. Последняя определяет начальные параметры окна: координаты окна на экране, его размер и рамку. Координаты и размер окна задаются как координаты верхнего левого угла окна и количество строк и столбцов в окне. Рамка окна задается как предопределенный указатель рамки окна.

Синтаксическая модель функции defs_wn:

```
defs_wn (&wn, rb, cb, row, col, &border);
```

```
WINDOW wn;      /* Структура, определяющая окно */
```

```
int rb, cb;      /* Координаты верхнего левого угла окна  
                  (строка и столбец) */
```



```
int row, col;      /* Размер окна (количество строк и
                   столбцов) */
BORDER border;    /* Структура, определяющая рамку окна */
```

Например, окно размером 10 строк на 20 столбцов и рамкой из одинарной линии, координата верхнего левого угла которого (10,25), определяется следующим образом:

```
WINDOW window;
defs_wn(&window,10,25,10,20,BDR_LNP);
```

Функция `def_wn` является альтернативной функцией определения начальных параметров окна. Она определяет координаты и размер окна в виде координат верхнего левого и нижнего правого углов окна.

Остальные параметры окна функции `defs_wn` и `def_wn` устанавливают по умолчанию.

Координаты и размер окна

Координаты окна определяют область экрана, в которой будет появляться окно, его размер — количество строк и столбцов (включая рамку окна) в окне. Координаты и размер окна задаются либо в виде координат верхнего левого угла окна и количества строк и столбцов в окне (функция `defs_wn`), либо в виде координат верхнего левого и нижнего правого углов окна (функция `def_wn`).

Изменить положение окна на экране и его размер после того, как они определены, позволяет функция `mod_wn`. Синтаксическая модель этой функции:

```
mod_wn (rb, cb, row, col, &wn);
int rb;      /* Новые координаты верхнего */
int cb;      /* левого угла окна */
int row;     /* Новый размер окна (количество */
int col;     /* строк и столбцов) */
WINDOW wn;   /* Структура, определяющая окно */
```

Если окно установлено на экране, то чтобы отразить изменение его положения и размера, необходимо окно переустановить, т.е. сначала удалить с экрана, а затем установить вновь.

Рамка окна

Окно может быть определено с рамкой или без рамки. Рамка окна задается при определении начальных параметров окна как параметр функции определения окна (`defs_wn` или `def_wn`), который может принимать следующие значения:

```
BDR_OP — рамка отсутствует,
BDR_LNP — рамка из одинарной линии,
BDR_DLNP — рамка из двойной линии,
BDR_DOTP — рамка из точек,
BDR_REVP — рамка инвертированного цвета.
```


Структура, определяющая рамку, имеет следующий формат:

```
struct BORDER {
    char bdr_att;    /* Атрибут цвета рамки */
    char hrz, vrt;   /* Символы, задающие горизонтальный
                     и вертикальный контур */
    char c1,c2,c3,c4; /* Символы, задающие углы рамки,
                     (начиная с верхнего левого
                     угла и далее по ходу
                     часовой стрелки) */
};
```

Например, рамка из одинарной линии определяется следующим образом:

```
struct BORDER bdr_ln={NORMAL,164,165,160,161,162,163};
#define BDR_LNP &bdr_ln
```

Предопределенная константа NORMAL определяет цвет рамки и означает обычный цвет для монохромного дисплея. Эта константа определяется как

```
#define NORMAL 0x07
```

Пользователь может определить свою рамку окна. Например, рамка из звездочек определяется следующим образом:

```
struct BORDER bdr_ast={RED,'*','*','*','*','*','*'};
#define BDR_ASTP &bdr_ast
```

Рамка будет иметь красный цвет.

Определение окна с данной рамкой:

```
defs_wn (&wn,rb,cb,row,col,BDR_ASTP);
```

Рамку окна можно изменить. Для этого предназначены функции `sw_border` и `v_border`. Функция `sw_border` изменяет рамку окна, но если окно уже установлено на экране, то на экране рамка не будет изменена. Чтобы вычертить новую рамку, когда окно уже установлено на экране, используется функция `v_border`.

Синтаксические модели этих функций:

```
sw_border (&border, &wn);
v_border (&wn, &border);
WINDOW wn;    /* Структура, определяющая окно */
BORDER border; /* Структура, определяющая новую рамку
                окна */
```

Тип окна

Поддерживаются два типа окон. Окно может быть либо простым, либо стековым. Стековое окно может появляться на экране и исчезать с экрана, не уничтожая содержимое той области экрана, на которой оно было установлено. Простое окно, наоборот, затирает область экрана, на которой оно устанавливается. Стековые окна используются,

например, при организации системы всплывающих меню. Простые окна используются в тех случаях, когда окно должно постоянно находиться на экране, например, для просмотра файлов.

По умолчанию окно определяется как простое окно.

Переключать тип окна позволяет функция `sw_wntyp`:

```
sw_wntyp (state, &wn);
WINDOW wn; /* Структура, определяющая окно */
int state; /* Переключатель типа окна */
```

Переключатель типа окна `state` принимает одно из двух значений:

`POPUP` — стековое окно;

`ORDINARY` — простое окно.

Имя окна

Окно может иметь имя, которое отображается на рамке в его верхнем левом углу, как показано на рис. 23.



Рис. 23. Окно с именем

По умолчанию окно имени не имеет. Функция `sw_name` позволяет назначить имя окну (до установки окна на экран):

```
sw_name (name, &wn);
WINDOW wn; /* Структура окна */
char * name; /* Указатель на строку, содержащую имя
              окна */
```

Например,

```
sw_name ("Окно ошибок", &wn);
```

Вызов функции `sw_name` вида:

```
sw_name (NULLP, &wn);
```

позволяет удалить присвоенное окну имя. `NULLP` — это нулевой указатель.

Если окно не имеет рамки, то имя окна не будет отображаться.

Рабочая область окна

Рабочая область окна — это область, в которую будет осуществляться вывод данных при записи их в окно. Рабочая область может занимать все окно или внутреннюю его часть, т.е. находиться в одном из следующих двух состояний:

FULL — рабочая область занимает полное окно, включая и рамку окна (если она есть);

INSIDE — рабочая область занимает не все окно; она не включает рамку окна и может быть отделена от левой и правой границ окна отступами любой ширины.

Окно, рабочая область которого находится в состоянии **INSIDE**, показано на рис. 24.

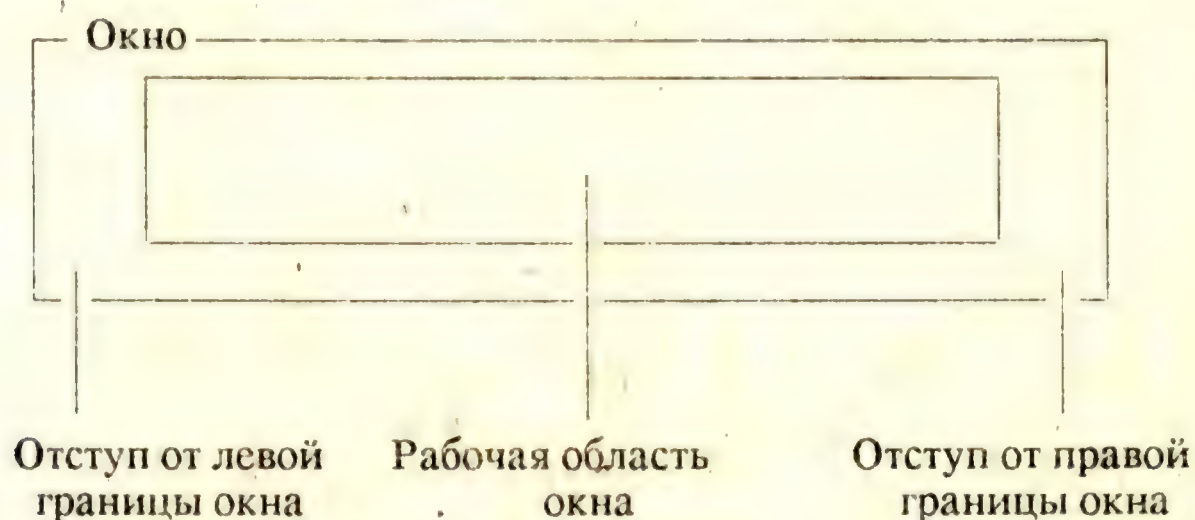


Рис. 24. Рабочая область окна

Функция определения окна `defs_wn` устанавливает ширину отступов от левой и правой границ окна равной 1. Альтернативная функция определения окна `def_wn` позволяет сразу определить величину отступов от левой и правой границ окна. Синтаксическая модель функции `def_wn`:

```
def_wn (&wn,rb,re,cb,ce,l_mg,r_mg,&border);
WINDOW wn;      /* Структура, определяющая окно */
int rb,cb;       /* Координаты верхнего левого угла окна */
int re,ce;       /* Координаты нижнего правого угла окна */
int l_mg;        /* Ширина отступа от левой границы */
int r_mg;        /* Ширина отступа от правой границы */
BORDER border;   /* Структура, определяющая рамку окна */
```

Например, определение окна размером 10 строк на 25 столбцов, с координатами верхнего левого угла (0,0), рамкой из двойной линии и рабочей областью, отделенной от левой и правой границ окна 5 пробелами, выглядит так:

```
WINDOW wn;
def_wn(&wn,0,9,0,24,5,5,BDR_DLNP);
```

До того как окно установлено на экране, переключатель рабочей области окна находится в состоянии **FULL** (область занимает все окно). При установке окна на экране рабочая область окна переключается в состояние **INSIDE**. Когда окно удаляется с экрана, переключатель рабочей области снова устанавливается в состояние **FULL**.

Функция `sw_margin` позволяет изменить размер рабочей области окна после того, как окно уже определено. Эта функция изменяет ширину отступов от левой и правой границ окна:


```
sw_margin(left_margin, right_margin, &wn);
WINDOW wn; /* Структура, определяющая окно */
int left_margin, /* Новая ширина отступов от */
right_margin; /* левой и правой границ окна */
```

Функция `dim_wn` позволяет установить для окна одно из двух состояний рабочей области:

```
dim_wn (state, &wn);
WINDOW wn; /* Структура, определяющая окно */
int state; /* FULL или INSIDE */
```

Цвет окна

С каждым символом, выводимым в окно, связывается атрибут цвета, который определяет, как и с каким цветом символ выводится в окно. Атрибут символа фактически определяет цвет текста в окне и цвет фона окна. Рамка окна также имеет атрибут цвета.

Для представления цветов предназначены две системы цветовых атрибутов: система физических атрибутов и система логических атрибутов.

Значения атрибута цвета называются физическими атрибутами. Физические атрибуты в виде предопределенных символических констант приведены в табл. 12.1. Атрибуты даны как для цветного, так и для монохромного адаптеров.

Таблица 12.1

Физические атрибуты				
Цветной дисплей		Монохромный дисплей		Категория атрибута
символическая константа	цвет	символическая константа	значение	
BLACK	Черный на синем	NORMAL	Обычный цвет	Базовый
BLUE	Синий на белом	UNDERLINE	Подчеркивание	
GREEN	Зеленый на синем	REVERSE	Инвертированный цвет	
CYAN	Голубой на синем			
RED	Красный на синем			
MAGENTA	Фиолетовый на синем			
BROWN	Коричневый на синем			
WHITE	Белый на синем			Модификатор, который может быть добавлен к одному из базовых атрибутов
LIGHT		HIGH_INT	Повышенная яркость	
		BLINK	Мерцание	

Если, например, при работе с системой физических атрибутов необходимо установить режим вывода в окно обычного текста повышенной яркости с мерцанием, необходимо задать следующее сочетание:

NORMAL+HIGH_INT+BLINK

Физические атрибуты использовать в программе не очень удобно, поскольку придется писать отдельный код для цветного и монохромного видеорежимов. Предпочтительнее использовать систему логических атрибутов. Логические атрибуты позволяют писать программы, которые могут использовать полную цветовую гамму цветного дисплея и будут совместимы с монохромным дисплеем.

Для каждого логического атрибута определены два физических атрибута: физический атрибут для цветного дисплея и физический атрибут для монохромного дисплея. При работе на цветном дисплее будет использоваться физический атрибут для цветного дисплея, а при работе на монохромном дисплее — физический атрибут для монохромного дисплея.

Логические атрибуты в виде предопределенных констант приведены в табл. 12.2.

Таблица 12.2

Логические атрибуты

Символическая константа	Цвет для монохромного дисплея	Цвет символа/фона для цветного дисплея	Применение атрибута
LDOS	Обычный	Белый/черный	Вывод Альфа-ДОС
LNORMAL	-"-	Белый/синий	Обычный текст
LHIGHLIGHT	Обычный повышенной яркости	Ярко-белый/синий	Акцентированный текст
LREVERSE	Инвертированный	Синий/белый	То же
LURGENT	Обычный повышенной яркости с мерцанием	Красный с мерцанием/черный	Текст повышенного внимания
LHELP	Обычный повышенной яркости	Синий/белый	Справочная информация
LERROR	Инвертированный	Красный/черный	Сообщения об ошибках
LMESSAGE	Обычный повышенной яркости	Ярко-белый/синий	Информационное сообщение
LMARK	Инвертированный	Синий/белый	Помеченный текст
LBLACK	-"-	Черный/синий	Для представления цвета текста
LBLUE	-"-	Синий/белый	
LGREEN	Обычный	Зеленый/синий	
LCYAN	-"-	Голубой/синий	
LRED	Инвертированный	Красный/синий	
LMAGENTA	-"-	Фиолетовый/синий	
LBROWN	Обычный	Коричневый/синий	
LWHITE	-"-	Белый/синий	

Пользователь может изменять физические атрибуты, связанные с предопределенными логическими атрибутами, и определить одну или более дополнительных систем логических атрибутов. Однако для большинства применений предоставляемая система логических атрибутов является вполне достаточной.

По умолчанию в программе устанавливается система логических атрибутов (определением символической константы `ATT_LOGIC` в файле включения `wios.h`). Чтобы ввести систему физических атрибутов, необходимо отменить определение этой символической переменной:

```
#include <wios.h>
#undef ATT_LOGIC    /* Ввод физических атрибутов */
```

Первоначально атрибуты цвета окна и рамки устанавливаются на `LNORMAL`, если используется система логических атрибутов, или `NORMAL`, если используется система физических атрибутов.

Для изменения атрибутов цвета окна и рамки окна предназначены следующие функции:

```
sw_att      — изменяет цвет символа;
color_wn    — изменяет цвет текста и фона окна;
sw_bdratt   — изменяет цвет рамки окна.
```

Функции `sw_att` и `sw_bdratt` работают как с логическими, так и с физическими атрибутами. Функция `color_wn` может использоваться, если программиста не устраивает предоставляемое сочетание цвета текста и цвета фона и работает только с физическими атрибутами. Синтаксические модели этих функций:

```
sw_att (attribute, &wn);
WINDOW wn;
int attribute;
/* Структура, определяющая окно */
/* Новый логический или физический
   атрибут цвета символа */

sw_bdratt (attribute, &wn);
WINDOW wn;
int attribute;
/* Структура, определяющая окно */
/* Новый логический или физический
   атрибут цвета рамки */

color_wn (foreground, background, &wn);
WINDOW wn;
char foreground;
char background;
/* Структура, определяющая окно */
/* Новый физический атрибут цвета
   текста */
/* Новый физический атрибут цвета
   фона */
```

Примечание. Изменять параметры окна нужно до установки окна на экране. В противном случае соответствующие изменения не будут отражены на экране.

12.2. ОПЕРАЦИИ С ОКНАМИ

Установка окна на экране и удаление с экрана

Полная процедура создания окна в программе должна включать следующие шаги:

1) включение файлов `wios.h` и `window.h`:

```
#include <wios.h>
#include <window.h>
```

Если в программе будет использоваться система физических атрибутов цветов, то между двумя директивами `#include` необходимо поместить директиву, отменяющую использование логических атрибутов:

```
#undef ATT_LOGIC
```

2) объявление окна:

```
WINDOW wn;
```

3) инициализация, необходимая перед работой с окном:

```
init_wns();
```

4) определение окна (определение начальных параметров окна) функцией `defs_wn` или `def_wn`;

5) изменение параметров окна, если необходимо;

6) установка окна на экране.

После того как окно установлено на экране, с окном можно выполнять всевозможные манипуляции: писать и читать информацию в окно и из окна, просматривать текстовые файлы в окне, перемещать его по экрану, сохранять образ окна в памяти и восстанавливать его на экране, удалять его с экрана и выполнять другие операции.

Для установки окна на экране используется функция `set_wn`:

```
set_wn (&wn);
WINDOW wn; /* Структура, определяющая окно */
```

Функция `set_wn` устанавливает как стековые, так и простые окна. Если окно является стековым, то функция `set_wn` сохраняет в памяти образ той части экрана, в которую окно будет помещено.

Окно удаляется с экрана функцией `unset_wn`:

```
unset_wn (&wn);
WINDOW wn;
```

Проверить, находится ли окно в данный момент на экране, позволяет функция `active_wn`:

```
active_wn (&wn);
WINDOW wn;
```


Например,

```
if (active_wn (&wn) == 1)  /* Окно установлено на
                           экране? */
```

...

Стековые окна могут накладываться одно на другое. Однако при этом следует соблюдать следующие правила:

- изменять информацию можно только в самом верхнем окне;
- удалять окна следует в порядке, обратном тому, в котором они помещались на экран.

Перемещение окон

Для перемещения окна по экрану предназначена функция `mv_wi`:

```
mv_wi (row, col, &wn);
WINDOW wn;  /* Структура, определяющая окно */
int row,col; /* Новые координаты окна на экране
              (строка и столбец) */
```

Использование этой функции для перемещения двух цветных окон по контуру экрана показано в примере.

Пример.

```
#include <wios.h>
#undef ATT_LOGIC  /* Ввод системы физических атрибутов */
#include <window.h>
#include <conio.h> /* Требуется для функции kbhit */
main()
{
    int i, j;
    /* Объявление окон */
    WINDOW wn1, wn2;
    /* Инициализация */
    init_wns();
    /* Определение окон */
    defs_wn(&wn1, 0, 0, 5, 10, BDR_OP);
    defs_wn(&wn2, 20, 70, 5, 10, BDR_OP);
    /* Установка окон как стековых */
    sw_wntype(POPUP, &wn1);
    sw_wntype(POPUP, &wn2);
    /* Установка цвета фона окон */
    color_wn(WHITE, WHITE, &wn1);
    color_wn(CYAN, CYAN, &wn2);
    /* Установка окон на экран */
    set_wn(&wn1);
    set_wn(&wn2);
    /* Цикл, пока не будет нажата клавиша */
    while (!kbhit())
    /* Перемещение окон по экрану */
        for (i = 1, j = 69; i < 70; i++, j--) {
            mv_wi(0, i, &wn1);
            mv_wi(20, j, &wn2);
```



```

    }
    for (i = 1, j = 19; i < 20; i++, j--) {
        mv_wi(i, 70, &wn1);
        mv_wi(j, 0, &wn2);
    }
    for (i = 70, j = 1; i != 0; i--, j++) {
        mv_wi(20, i, &wn1);
        mv_wi(0, j, &wn2);
    }
    for (i = 20, j = 1; i != 0; i--, j++) {
        mv_wi(i, 0, &wn1);
        mv_wi(j, 70, &wn2);
    }
}
/* Удаление окон с экрана */
unset_wn(&wn1);
unset_wn(&wn2);
}

```

Пересылка окна между экраном и памятью

Образ окна (включая и рамку окна) можно сохранить в памяти, а затем восстанавливать его на экране. Для этого предназначены следующие функции:

`sav_wi` — сохраняет образ окна в памяти;

`unsav_wi` — восстанавливает на экране ранее сохраненный образ окна и не оставляет образ в памяти;

`gerl_wi` — восстанавливает на экране ранее сохраненный образ окна и оставляет образ в памяти.

Пересылка окна между экраном и файлом на диске

Образ окна можно с помощью функций ввода-вывода библиотеки Си сохранять в файле на диске, а затем восстанавливать его на экране.

Сохранение образа окна в файле на диске можно выполнить следующей последовательностью действий:

1) получить размер окна:

```
size = size_wn(FULL, WIND, &wn);
```

Функция `size_wn` возвращает количество байтов в рабочей области окна. Для каждой позиции окна отводятся два байта: один — для символа, а другой — для атрибута символа. Первый параметр функции указывает, какой из двух размеров рабочей области окна (`FULL` или `INSIDE`) используется для вычисления, а второй параметр определяет часть окна, размер которой вычисляется. Символическая константа `WIND` означает, что вычисляется размер полного окна;

2) сохранить образ окна в памяти:

```
sav_wi(&wn);
```

Функция `sav_wi` сохраняет образ окна с экрана в памяти в видеост-

роке. Видеострока не является стандартной строкой Си. Видеострока — это массив символов, предназначенный для сохранения информации с экрана, т.е. пар "символ — атрибут". Нулевой символ не добавляется в конец видеостроки. Адрес видеостроки сохраняется в структуре, определяющей окно. Окно сохраняется полностью, включая и рамку окна;

3) открыть файл на диске и записать образ окна из памяти в файл:

```
fp = fopen (...);
fwrite(get_image(&wn), size, 1, fp);
```

Функция `get_image` возвращает указатель на строку, содержащую образ окна.

Восстановление образа окна на экране из файла на диске можно выполнить следующей последовательностью действий:

1) получить память для образа окна:

```
image = get_mem (size);
```

Функция `get_mem` получает память заданного размера и возвращает указатель на распределяемую память;

2) открыть файл на диске и прочитать из него образ окна:

```
fp = fopen (...);
fread(image, size, 1, fp);
```

3) связать строку, содержащую образ окна, со структурой, определяющей окно:

```
sav_image(image, &wn);
```

4) восстановить образ окна из памяти на экране:

```
unsave_wi(&wn);
```

Функция `unsave_wi` помещает на экран содержимое окна, сохраненное в строке, связанной со структурой окна, и освобождает память, используемую для сохранения.

Операции ввода-вывода для окна

После того как окно определено (объявлено и инициализировано), для него можно выполнять операции ввода и вывода. При выполнении операций ввода и вывода используется элемент окна, называемый "виртуальным курсором окна". Виртуальный курсор окна определяет позицию в окне, начиная с которой будет выполняться чтение из окна или запись в окно. Виртуальный курсор окна не виден на экране и не зависит от положения физического курсора экрана. Для каждого окна определяется свой виртуальный курсор. Координаты виртуального курсора окна содержатся в структуре типа `WINDOW`, определяющей окно. Координаты точки, от которой отсчитывается положение виртуального курсора, зависят от того, находится рабочая область окна в

состоянии FULL или INSIDE. Если рабочая область окна находится в состоянии FULL (занимает все окно), то положение виртуального курсора определяется относительно верхнего левого угла (верхнего левого угла рамки, если окно с рамкой). Если же рабочая область окна находится в состоянии INSIDE (занимает не все окно), то положение виртуального курсора задается относительно первой позиции рабочей области окна.

После того как рабочая область окна определена, координаты точки, от которой задается положение виртуального курсора, не меняются.

Первоначально виртуальный курсор имеет координаты (0,0) (верхний левый угол окна, если рабочая область окна в состоянии FULL, или первая позиция рабочей области, если рабочая область в состоянии INSIDE).

Для перемещения виртуального курсора окна в позицию, с которой будет осуществляться операция ввода-вывода, используется функция `mv_cs`:

```
mv_cs(row, col, &wn);  
int row,col;      /* Новая позиция виртуального курсора в  
                  окне */  
WINDOW wn; /* Структура, определяющая окно */
```

Позиция курсора экрана не зависит от позиции виртуального курсора окна и не меняется (по умолчанию) при выполнении операций ввода-вывода для окна. Для управления курсором окна используются следующие функции:

`mv_csr(row, col, &wn)` — изменяет положение виртуального курсора и помещает курсор экрана в позицию виртуального курсора;

`pl_csr(&wn)` — помещает курсор экрана в позицию виртуального курсора окна.

Вывод в окно

Данные можно выводить как в окно, установленное на экране, так и в окно, не установленное на экране. На операции вывода данных в окно влияет установка следующих режимов вывода в окно:

- перемещение виртуального курсора окна. Режим перемещения виртуального курсора определяет, надо ли изменять позицию виртуального курсора окна при выполнении операции вывода в окно. Если этот режим разрешен, то после выполнения операции вывода в окно виртуальный курсор будет помещен в позицию после последнего записанного символа. В противном случае позиция виртуального курсора не будет изменяться;

- прокрутка текста окна. Режим прокрутки текста окна определяет, поддерживается или нет автоматическая прокрутка текста окна при выводе в заполненное окно. Если этот режим разрешен, то при

выводе в заполненное окно содержимое окна будет прокручиваться вверх на одну строку и вывод будет продолжаться в последнюю строку окна. В противном случае вывод будет прекращаться при заполнении окна;

- перенос целого слова. Режим переноса целого слова указывает, поддерживается или нет перенос целого слова, если вывод не помещается в строку окна. Если перенос целого слова разрешен, то слова не будут разрываться при переносе. В противном случае вывод будет полностью заполнять строку окна вне зависимости от того, разрывается слово или нет;

- очистка конца строки окна. Режим очистки конца строки окна управляет очисткой позиций, начиная от символа после последнего записанного и кончая концом строки окна. Это равносильно удалению предыдущего содержимого строки окна перед записью в нее. В большинстве случаев автоматическая очистка конца строки окна полезна. Автоматическая очистка конца строки нежелательна при обновлении только небольшой части строки окна, при котором оставшаяся часть строки окна не должна изменяться.

Для переключения режимов вывода используются следующие функции:

`sw_csadv(switch, &wn)` — управление перемещением виртуального курсора окна;

`sw_scrall(switch, &wn)` — управление прокруткой текста окна;

`sw_wwrap(switch, &wn)` — управление переносом целого слова;

`sw_clear(switch, &wn)` — управление очисткой конца строки окна.

Переключатель `switch` может принимать одно из двух значений:

`ON` — режим включен;

`OFF` — режим отключен.

Для окна можно выполнять следующие разновидности операций вывода:

1) вывод строки Си. Вывод строки Си в окно выполняют следующие функции: `v_st`, `v_fst`, `v_plst`. Синтаксические модели этих функций:

```
v_st (string, &wn);
```

```
v_fst (string, &wn);
```

```
v_plst (row, col, string, &wn);
```

```
WINDOW wn;
```

```
char * string;    /* Строка Си, выводимая в окно */
```

```
int row, col;     /* Позиция окна для вывода */
```

Функция `v_st` осуществляет вывод строки Си в окно, начиная с позиции, задаваемой виртуальным курсором окна; она прекращает вывод при заполнении окна. Если разрешена прокрутка текста в окне и окно уже заполнено, то функция `v_st` вызовет прокрутку вверх на одну строку и запись в последнюю строку окна. Если выводимая стро-

ка длиннее, чем ширина окна, то вывод прекращается при заполнении окна. Функция `vfst` идентична функции `vst`, но гарантирует то, что вся строка попадет в окно, даже если она превышает ширину окна. Функция `vplst` обеспечивает вывод строки в заданную позицию окна.

Функция `vplst` может выполнять центрирование текста в окне и выравнивание его к правой или левой границе окна. Для этого в качестве аргумента `col` необходимо задать одну из следующих символических констант:

`CENTER_TXT` — центрирование текста в строке окна;

`RIGHT_TXT` — выравнивание текста к правой границе окна;

`LEFT_TXT` — выравнивание текста к левой границе окна;

2) вывод форматизованных данных (функция `vprintf`);

3) вывод символа (функция `vch`);

4) изменение атрибута символа (функция `vatt`);

5) прокрутка содержимого окна. Прокрутку содержимого окна выполняет функция `mv_rws`:

```
mv_rws(n, dir, &wn);
```

```
WINDOW wn;
```

```
int n;      /* Число прокручиваемых строк */
```

```
char dir;   /* Направление прокрутки */
```

Направление прокрутки задается константами `UP` (прокрутка вверх) и `DOWN` (прокрутка вниз).

Если требуется прокрутка не всего окна, а какой-то его части, можно определить отдельное окно, наложить его на ту часть, где требуется прокрутка, и затем использовать функцию `mv_rws` для этого отдельного окна;

6) очистка окна (функция `cl_wn`).

Чтение из окна

Обеспечиваются следующие операции чтения из окна:

1) чтение символа (функция `vo_ch`);

2) чтение атрибута символа (функция `vo_att`);

3) чтение части окна. Имеются две функции `v_mova` и `v_mov`, которые позволяют читать какую-то определенную часть окна. В общем случае эти функции позволяют делать несколько больше: они дают возможность пересылать данные между окном и памятью.

Синтаксические модели этих функций:

```
v_mov(vst, &wn, part, dir);
```

```
v_mova(st, &wn, part, dir);
```

```
WINDOW wn;      /* Структура, определяющая окно */
```

```
char vst[];      /* Видеострока */
```

```
char st[];       /* Строка Си */
```

```
int part;        /* Часть окна для пересылки */
```

```
int dir;         /* Направление пересылки */
```


Отличие между этими функциями заключается в том, что функция `v_tova` пересылает данные между окном и строкой Си (т.е. пересылает только символьную информацию окна), а функция `v_tov` — пересылает данные между окном и видеострокой (т.е. пересылает и символы, и атрибуты символов).

Можно пересылать следующие части окна (значения, которые принимает аргумент `part`):

`CH` — символ в позиции виртуального курсора;

`ROW` — строка, в которой расположен виртуальный курсор;

`COL` — столбец, в котором расположен виртуальный курсор;

`ENDROW` — от текущей позиции виртуального курсора до конца строки;

`ENDCOL` — от текущей позиции виртуального курсора до конца столбца;

`ENDWIND` — от строки, в которой расположен виртуальный курсор, до конца окна;

`WIND` — целое окно.

Направление пересылки (аргумент `dir`):

`IN` — с экрана в строку;

`OUT` — из строки на экран;

4) чтение окна в файл на диске. Функция `copy_wc` копирует символьную информацию окна в файл. Можно копировать все окно (включая и рамку) либо рабочую область окна:

```
copy_wc(dim, filename, mode, &wn);
WINDON wn;      /* Структура, определяющая окно */
char dim;        /* FULL или INSIDE */
char *filename;  /* Имя файла */
char *mode;      /* Режим доступа к файлу */
```

12.3. ПРОСМОТР ФАЙЛА В ОКНЕ

Функции работы с окнами библиотеки Си позволяют организовать просмотр текстовых файлов в окне. Для этого необходимо подготовить каким-либо редактором текстовый файл на диске, прочитать содержимое файла с диска в файл памяти и отобразить файл памяти в окно с организацией прокрутки файла в окне.

Создание файла в памяти

Для создания файла в памяти, в который будет помещаться текстовый файл с диска, необходимо выполнить следующее:

- объявить файл в памяти;
- связать его с файлом на диске;
- поместить информацию файла на диске в файл памяти.

Информация, необходимая для управления и отображения файла

памяти, хранится в структуре типа `FREC`, которая описывает файл памяти. Объявление файла памяти имеет вид:

```
FREC mem_file;
```

Для связи файла памяти с файлом на диске используется функция `def_fr`:

```
def_fr(&mem_file, file, lines, cols);  
FREC mem_file;      /* Структура, описывающая файл памяти */  
char *file;          /* Строка, содержащая имя файла на диске */  
int lines;           /* Число строк в файле памяти */  
int cols;            /* Максимальное число позиций в строке, */  
                    /* не включая символы конца строки и */  
                    /* новой строки */
```

Если памяти для создания файла недостаточно, функция `def_fr` возвращает ненулевое значение (в противном случае — ноль).

Для чтения в файл памяти из файла на диске предназначена функция `di_file`:

```
di_file(&mem_file);  
FREC mem_file;
```

Если файл на диске, из которого читаются данные в файл памяти, превышает размер файла в памяти, то последние строки не будут прочитаны, поэтому конец текстового файла нельзя будет просмотреть в окне. Если размер строки файла на диске больше максимального размера строки файла памяти, то в файл памяти будет помещаться усеченная строка.

Просмотр файла памяти в окне

Для организации просмотра файла памяти в окне необходимо:

- связать файл памяти с окном, в котором он будет отображаться;
- отобразить файл памяти в окне и организовать прокрутку файла памяти в окне.

Для того чтобы связать файл памяти с окном, в котором он будет отображаться, предназначена функция `sw_mfile`:

```
sw_mfile (&mem_file, &wn);  
FREC mem_file;  
WINDOW wn;
```

После ее выполнения необходимо установить окно на экране и отобразить в него файл из памяти. Это выполняет функция `vs_file`:

```
vs_file(exit_key, &wn);  
WINDOW wn;  
int exit_key; /* Клавиша для окончания просмотра  
              файла в окне */
```

Функция `vs_file` позволяет прокручивать файл памяти в окне, ис-

пользуя клавиши управления курсором. Нажатие клавиши `exit_key` вызовет прекращение просмотра файла и удаление окна с экрана.

Список клавиш, используемых для прокрутки файла в окне, приведен ниже.

Клавиши управления прокруткой файла в окне

Клавиша	Функция
→ ← ↑ ↓	Прокрутка на одну позицию в направлении стрелки
СТР↑	Прокрутка на окно вверх
СТР↓	Прокрутка на окно вниз
НАЧ	Прокрутка в начало файла
КОН	Прокрутка в конец файла

Если текстовый файл полностью помещается в окне и нет необходимости в его прокрутке, то для отображения файла в окно вместо функции `vs_file` можно использовать функцию `v_file`. Она может быть использована для организации собственного цикла прокрутки файла в окне (совместно с функциями чтения с клавиатуры и функцией `k_vcom`, которая используется для преобразования кодов клавиш перемещения курсора, возвращаемых функциями чтения с клавиатуры, в команды управления прокруткой файла памяти в окне).

Следующий пример представляет программу просмотра текстового файла в окне. Имя просматриваемого файла является параметром программы. Окно занимает весь экран и имя просматриваемого файла отображается на рамке в верхнем левом углу окна.

Пример.

```
/* Эта программа позволяет просмотреть текстовый файл в
окне. Имя файла передается как аргумент при вызове */
#include <wios.h>      /* Требуется для объявления функций */
#include <window.h>    /* работы с окнами */
#define EXIT_KEY      K_ESC
#define MAXLINES      1000
#define MAXCOLS        80
char name[30] = "Файл ";
main(int argc, char*argv[])    /* В функцию main передается
                                аргумент */
{
    WINDOW wn;                /* Объявление окна для просмотра */
    FREQ mf;                   /* Объявление файла памяти */
    char *ptr = argv[1];
    int i;
    init_wns();                /* Инициализация */
    defs_wn(&wn, 0, 0, 24, 79, BRD_LNP); /* Определение окна */
    sw_wntyp(POPUP, &wn);      /* Окно стековое */
    for (i = 5; name[i] = *(argv[1]++); i++); /* Формирование
                                                имени окна */
    sw_name(name, &wn);
    /* Создание файла памяти, связывание его с файлом на
    диске и чтение файла с диска в файл памяти */
    if (!def_fr(&mf, ptr, MAXLINES, MAXCOLS))
```



```
errout("Нет памяти для файла ", ptr);
if (!di_file(&mf))
    errout("Ошибка чтения файла ", ptr);
sw_mfile(&mf, &wn);
vs_file(EXIT_KEY, &wn); /* Просмотр файла в окне */
                          /* Нажатие клавиши Выход означает */
                          /* конец просмотра и удаление */
                          /* окна с экрана */
free_file(&mf);          /* Освободить память, занимаемую */
return (0);              /* файлом памяти */
}
```

В программе используется функция `errout`, которая предназначена для выхода из программы при возникновении ошибки с выводом сообщения об ошибке в последней строке экрана. Сообщение состоит из двух строк. Обычно одна строка является фиксированным сообщением, а вторая содержит переменную информацию, например, имя файла.

12.4. СОЗДАНИЕ ОКОННЫХ МЕНЮ

Функции работы с окнами библиотеки Си могут быть использованы для разработки прикладных систем, использующих оконные меню.

Оконные меню представляют собой иерархию стековых окон, которые появляются на экране при выборе некоторого действия на более высоком уровне, например, из главного меню, и содержат список команд, входящих в данное меню.

Из появившегося меню выбирается нужная команда, после чего меню исчезает с экрана и выполняется выбранная команда меню.

В библиотеке Си имеется функция с именем `rmenu`, которая упрощает создание оконных меню в программе. Эта функция позволяет устанавливать predetermined меню на экран, перемещаться с помощью клавиш управления курсором от одного элемента меню к другому с высвечиванием более ярким светом текущего элемента и возвращать число, указывающее на выбранный элемент меню. После выбора требуемого элемента меню исчезает с экрана и восстанавливается первоначальный образ экрана. Элементы в меню могут располагаться вертикально или горизонтально.

До вызова функции `rmenu` необходимо подготовить информацию меню и создать окно вывода. Для этого необходимо:

- создать файл меню в памяти;
- создать окно для меню и связать его с файлом меню.

Создание файла меню в памяти

Для выполнения функции `rmenu` требуется, чтобы выдаваемое ею меню находилось в файле памяти. Файл памяти с меню можно получить, создав файл текстовым редактором и прочитав его (функцией

di_file) в файл памяти; можно также создать меню прямо в программе (функцией sti_file).

Формат меню должен удовлетворять следующим требованиям:

- меню может содержать любое число элементов;
- элементы меню могут располагаться в одной или более строках и каждая строка может содержать один или более элементов. В каждой строке (кроме последней) должно быть одно и то же число элементов;
- каждый элемент должен иметь одинаковую длину. Если элементы имеют неодинаковую длину, то можно либо дополнить элемент пробелами до нужной длины, либо отцентрировать его в строке пробелами.

Пример создания меню прямо в программе:

```
FREC menu_file;
def_fr (&menu_file, NULLP, 6, 18, &menu_file);
sti_file ("Новый файл", 0, &menu_file);
sti_file ("Открыть файл", 1, &menu_file);
sti_file ("Слить файлы", 2, &menu_file);
sti_file ("Сохранить файл", 3, &menu_file);
sti_file ("Сохранить с именем", 4, &menu_file);
sti_file ("Печатать файл", 5, &menu_file);
```

Нулевой указатель (NULLP) вместо имени файла в функции def_fr указывает на то, что файл памяти не связан с файлом на диске. Файл памяти создается вызовами функции sti_file, которая помещает строку в файл памяти. В этом случае функция di_file не используется.

Создание окна для меню

Окно для меню может быть такого же размера, как и файл меню, либо иметь меньший размер. В последнем случае не все элементы меню будут видны и для выбора элемента файл меню будет прокручиваться. При задании окна для меню используется одна из функций определения окна: defs_wn или def_wn.

Например, окно для файла меню, созданного выше, определяется функциями:

```
defs_wn(&menu_wn, 1, 0, 8, 20, BDR_LNP);
sw_margin (0, 0, &menu_wn);
```

Для связи окна с файлом меню в памяти используется функция sw_mfile:

```
sw_mfile(&menu_file, &menu_wn);
```

Вызов меню

После того как созданы файл меню в памяти и окно для него, вызывается функция rmenu для установки меню на экран:

```
int rmenu(&menu_wn, q, len, qr, init);
WINDOW menu_wn; /* Структура, определяющая окно для
                  меню */
int q; /* Число элементов в меню */
```



```
int len;          /* Длина элементов */
int qr;           /* Число элементов в строке */
int init;         /* Номер элемента (начиная с 1), в
                  /* который первоначально помещается
                  /* световая строка */
```

Функция `rmenu` возвращает номер выбранного элемента, который можно сохранить для последующего использования.

Выбор элемента осуществляется клавишами перемещения курсора и нажатием клавиши ВВОД. После выбора элемента меню исчезает с экрана.

12.5. СОЗДАНИЕ СПРАВОЧНОЙ СИСТЕМЫ

Справочная информация должна появляться на экране, когда пользователю необходимо получить подсказку о работе системы. Справочная система использует стековое окно, в котором просматривается текстовый файл справочной информации по той же схеме, по которой производится просмотр обычного текстового файла в окне. Для создания справочной системы необходимо:

- создать файл памяти, в который будет помещаться файл справочной информации с диска;
- прочитать файл справочной информации с диска в файл памяти;
- организовать просмотр файла справочной информации в окне.

Следующий пример показывает реализацию функции `help`, которая будет вызываться в программе, когда пользователь нажимает, например, клавишу F1, чтобы получить справочную информацию. В функции `help` предполагается, что файл справочной информации организован в виде последовательности экранов (или кадров) по 18 строк в каждом. Просмотр экранов справочной информации управляют с помощью трех командных кнопок: "Вперед", "Назад" и "Снять". Выбор командной кнопки "Вперед" означает листание на один экран справочной информации вперед, кнопки "Назад" — листание на один экран справочной информации назад, кнопки "Снять" — удаление справки с экрана и восстановление предыдущего содержимого экрана. Выбор командной кнопки производится нажатием клавиши ВВОД. Активная командная кнопка, т.е. та, которая выбирается при нажатии клавиши ВВОД, выделяется двойной линией. Переключение активной командной кнопки производится клавишами "→" и "←". Экран дисплея в момент работы функции `help` показан на рис. 25.

Для организации просмотра справочной информации в функции `help` используются пять окон: одно большое окно, занимающее весь экран дисплея, на которое накладываются окно просмотра экранов справочной информации и три окна для командных кнопок.

Для просмотра экранов справочной информации в окне просмотра не используется функция `vs_file`, а организован цикл просмотра.

Имя меню	Команды	Набор клавиш
Экран	Экран вывода	Ф4
Где	Увеличить	УПР+Ф10
	Заданный текст	УПР+\
	Следующий повтор	Ф3
Запуск	Ошибка ниже	РЕГ+Ф3
	Ошибка выше	РЕГ+Ф4
	Рестарт	РЕГ+Ф5
	Пуск	Ф5
	До курсора	Ф7
Отладка	Шаг трассы	Ф8
	Шаг программы	Ф10
	Точка прерывания	Ф9
	Тема	Ф1
	О справке	РЕГ+Ф1
Справка		



Рис. 25. Экран справочной системы

Для чтения с клавиатуры используется функция `ki`. Она переходит в состояние ожидания до тех пор, пока не будет нажата клавиша, и возвращает код нажатой клавиши. Как отрицательные значения возвращаются коды следующих клавиш и комбинаций клавиш:

- функциональных клавиш;
- клавиш малой цифровой клавиатуры (клавиш управления курсором);
- УПР+клавиша перемещения курсора;
- АЛТ+буквенно-цифровая клавиша.

Файл справочной информации отображает в окне функция `v_file`, а функция `pl_mfun` изменяет координаты окна в файле (устанавливает окно просмотра на нужный экран справочной информации).

Функция `v_file` только отображает файл в окне. Функция `vs_file`, в отличие от функции `v_file`, устанавливает окно просмотра на экране, отображает файл в окне, управляет прокруткой файла в окне с помощью клавиш перемещения курсора и удаляет окно с экрана при нажатии клавиши завершения просмотра.

Пример.

```

#include <wios.h> /* Требуется для функций */
#include <window.h> /* поддержки окон и содержат */
/* прототипы этих функций */
#define ENTER_KEY 13 /* Коды клавиш, возвращаемые */
#define ESC_KEY 27 /* функциями чтения с кла- */
#define LEFT_KEY -75 /* виатуры. Коды специальных */
#define RIGHT_KEY -77 /* клавиш возвращаются как */
/* отрицательные значения */
#define COLS 76 /* Длина строки файла */
/* В функцию help передаются имя файла справочной информации
 * на диске и количество строк в файле
 */
int help(char* file, int file_len)
{
    /* Объявление окон */
    WINDOW big_wn, /* Окно, на которое накладываются осталь- */
                  /* ные окна. Занимает весь экран */
        help_wn, /* Окно, в котором отображается подсказка */
        err_wn, /* Окно сообщения об ошибке */
        but_wn1, /* Три окна для командных кнопок */
        but_wn2,
        but_wn3;
    FREC help_file; /* Объявление файла памяти, в который */
                  /* будет читаться файл справочной */
                  /* информации с диска */
    int button; /* Номер активной командной кнопки */
    int key; /* Код клавиши */
    int fr = 0, fc = 0; /* Координаты окна подсказки в файле */
                  /* памяти */
    init_wns(); /* Инициализация системы работы с */
                  /* окнами */
    defs_wn(&big_wn, 0, 0, 25, 80, BDR_LNP); /* Определение */
                  /* большого окна */
    sw_wntyp(POPUP, &big_wn); /* Окно имеет обычные */
    sw_att(LNORMAL, &big_wn); /* атрибуты и имя "СПРАВКА" */
    sw_bdratt(LNORMAL, &big_wn);
    sw_name("СПРАВКА", &big_wn);
    defs_wn(&help_wn, 2, 2, 18, 76, BDR_0P); /* Определение */
                  /* окна подсказки */
    sw_wntyp(POPUP, &help_wn); /* Окно не имеет рамки, имеет */
    sw_att(LHELP, &help_wn); /* атрибут цвета LHELP и */
    dim_wn(FULL, &help_wn); /* рабочая область занимает */
                  /* все окно */
    defs_wn(&err_wn, 10, 28, 5, 25, BDR_LNP); /* Определение */
                  /* окна ошибки */
    sw_wntyp(POPUP, &err_wn); /* Окно имеет атрибут цвета */
    sw_att(LURGENT, &err_wn); /* LURGENT (мигающее окно) */
    sw_bdratt(LURGENT, &err_wn);

```



```

defs_wn(&but_wn1, 21, 9, 3, 10, BDR_LNP); /* Определение */
sw_wntyp(POPUP, &but_wn1); /* окна для командной кнопки */
/* "Вперед" */

sw_att(LHELP, &but_wn1);
sw_bdratt(LHELP, &but_wn1);
defs_wn(&but_wn2, 21, 34, 3, 10, BDR_LNP); /* Определение */
sw_wntyp(POPUP, &but_wn2); /* окна для командной кнопки */
/* "Назад" */

sw_att(LHELP, &but_wn2);
sw_bdratt(LHELP, &but_wn2);
defs_wn(&but_wn3, 21, 59, 3, 10, BDR_LNP); /* Определение */
sw_wntyp(POPUP, &but_wn3); /* окна для командной кнопки */
/* "Снять" */

sw_att(LHELP, &but_wn3);
sw_bdratt(LHELP, &but_wn3);
def_fr(&help_file, file, file_len, COLS); /* Создание файла */
sw_mfile(&help_file, &help_wn); /* памяти для файла спра- */
/* вочной информации и */
/* связывание его с окном */
/* подсказки */

/* Чтение файла с диска в файл памяти. Если при чтении
 * происходит ошибка, выдать окно с сообщением об ошибке и
 * завершить функцию после нажатия пользователем клавиши Вых
 */
if (!di_file(&help_file)) {
    set_wn(&err_wn);
    v_plst(1, CENTER_TXT, "Нет файла подсказки", &err_wn);
    while (ki() != ESC_KEY);
    unset_wn(&err_wn);
    return(-1);
}
set_wn(&big_wn); /* Установить большое окно */
/* Наложить на него три командных */
/* кнопки и окно просмотра спра- */
/* вочной информации */

set_wn(&but_wn1);
v_plst(1, CENTER_TXT, "Вперед", &but_wn1);
set_wn(&but_wn2);
v_plst(1, CENTER_TXT, "Назад", &but_wn2);
set_wn(&but_wn3);
v_plst(1, CENTER_TXT, "Снять", &but_wn3);
set_wn(&help_wn);
v_file(&help_wn); /* Отобразить в окне просмотра */
/* первый экран справочной ин- */
/* формации */

v_border(&but_wn1, BDR_DLNP); /* Отрастить рамкой из двой- */
/* ной линии, что первона- */
/* чально активной является */
/* кнопка "Вперед" */

```



```
/* Цикл просмотра файла справочной информации, пока
 * пользователь не выберет командную кнопку "Снять"
 */
while (1) {
    key = ki();
    /* Ожидание нажатия клавиши */
    switch (key) {
        /* Нажата клавиша "→" */
        case RIGHT_KEY: switch (button) {
            /* Активна командная кнопка "Вперед" — */
            /* сделать активной кнопку "Назад" */
            case 1: v_border(&but_wn1, BDR_LNP);
                    v_border(&but_wn2, BDR_DLNP);
                    button = 2;
                    break;

            /* Активна командная кнопка "Назад" — */
            /* сделать активной кнопку "Снять" */
            case 2: v_border(&but_wn2, BDR_LNP);
                    v_border(&but_wn3, BDR_DLNP);
                    button = 3;
                    break;

            /* Активна командная кнопка "Снять" — */
            /* сделать активной кнопку "Вперед" */
            case 3: v_border(&but_wn3, BDR_LNP);
                    v_border(&but_wn1, BDR_DLNP);
                    button = 1;
                    break;
        }
        break;

        /* Нажата клавиша "←" */
        case LEFT_KEY: switch (button) {
            /* Активна командная кнопка "Вперед" — */
            /* сделать активной кнопку "Снять" */
            case 1: v_border(&but_wn1, BDR_LNP);
                    v_border(&but_wn3, BDR_DLNP);
                    button = 3;
                    break;

            /* Активна командная кнопка "Назад" — */
            /* сделать активной кнопку "Вперед" */
            case 2: v_border(&but_wn2, BDR_LNP);
                    v_border(&but_wn1, BDR_DLNP);
                    button = 1;
                    break;

            /* Активна командная кнопка "Снять" — */
            /* сделать активной кнопку "Назад" */
            case 3: v_border(&but_wn3, BDR_LNP);
                    v_border(&but_wn2, BDR_DLNP);
                    button = 2;
                    break;
        }
        break;
    }
}
```



```

/* Нажата клавиша ВВОД */
case ENTER_KEY: switch (button) {
    /* Активна командная кнопка "Вперед" — */
    /* пролистать файл справочной информации */
    /* на один экран вперед. Выдать звуковой */
    /* сигнал при попытке выхода за пределы */
    /* файла */
    case 1: if (fr + 18 < file_len) {
        fr += 18;
        pl_mfwn(fr, fc, &help_wn);
        v_file(&help_wn);
    }
    else
        bell();
    break;

    /* Активна командная кнопка "Назад" — */
    /* пролистать файл справочной информации */
    /* на один экран назад. Выдать звуковой */
    /* сигнал при попытке выхода за пределы */
    /* файла */
    case 2: if (fr - 18 >= 0) {
        fr -= 18;
        pl_mfwn(fr, fc, &help_wn);
        v_file(&help_wn);
    }
    else
        bell();
    break;

    /* Активна командная кнопка "Снять" — */
    /* удалить все окна в порядке, обратном тому, */
    /* в котором они устанавливались на экран, */
    /* освободить память, занимаемую файлом */
    /* памяти, и завершить функцию */
    case 3: unset_wn(&help_wn);
        unset_wn(&but_wn1);
        unset_wn(&but_wn2);
        unset_wn(&but_wn3);
        unset_wn(&big_wn);
        free_file(&help_file);
        return(0);
    }
    break;

/* Нажата любая другая клавиша — выдать звуковой сигнал и */
/* ничего не делать */
default: bell();
    break;
}
}
}

```


Глава 13

ГРАФИКА

Для построения графических изображений на экране дисплея в библиотеку функций Си включен набор графических функций, которые могут быть вызваны из программ на языке Си, а также из программ на других языках, в которых поддерживаются соглашения о связях с программами на языке Си. Для использования функций графической библиотеки оборудование ПЭВМ должно поддерживать растровую графику.

Растровая графика базируется на понятии пикселя — наименьшей адресуемой точки на экране видеомонитора.

13.1. СТРУКТУРА ГРАФИЧЕСКОЙ ПРОГРАММЫ

Любая графическая Си-программа организуется по следующей схеме:

- установка видеорежима;
- создание и манипулирование графическими объектами;
- восстановление первоначальной видеоконфигурации перед выходом из программы.

В Си-программу, использующую графические функции, должен быть включен (директивой `#include`) файл `graph.h`. Файл `graph.h` содержит объявления структур и символических констант, используемых графическими функциями, а также прототипы графических функций. Перечень графических функций с указанием их прототипов приведен в приложении 2.

Установка видеорежима

Первым шагом любой графической программы является установка видеорежима, обеспечивающего требуемые графические операции. Видеорежим определяет размер экрана в пикселях и количество допустимых цветов.

Процесс установки видеорежима происходит следующим образом. Сначала необходимо попытаться установить режим с наивысшей разрешающей способностью. Если начальный режим установить нельзя, следует попытаться установить режим с меньшими требованиями к разрешающей способности до тех пор, пока режим не будет установлен, либо пока не станет ясно, что режим с требуемой для программы разрешающей способностью установить нельзя. В последнем случае выйти из программы и выдать сообщение об ошибке.

Видеорежим устанавливается функцией `_setvideomode`:

```
_setvideomode(short videomode);
```


Параметр `videomode` задается в виде predetermined символической константы. Константы и соответствующие им видеорежимы приведены в табл 13.1.

Таблица 13.1

Видеорежимы

Режим	Тип ¹	Размер ²	Количество цветов	Адаптер
<code>_DEFAULTMODE</code>	Восстанавливает экран к исходному состоянию			
<code>_TEXTBW40</code>	МТ	40*25	16 оттенков серого цвета	CGA
<code>_TEXTC40</code>	ЦТ	40*25	16	CGA
<code>_TEXTBW80</code>	МТ	80*25	16 оттенков серого цвета	CGA
<code>_TEXTC80</code>	ЦТ	80*25	16	CGA
<code>_MRES4COLOR</code>	ЦГ	320*200	4	CGA
<code>_MRESNOCOLOR</code>	МГ	320*200	4 оттенка серого цвета	CGA
<code>_HRESBW</code>	МГ	640*200	2	CGA
<code>_TEXTMONO</code>	МТ	80*25	1	MA
<code>_MRES16COLOR</code>	ЦГ	320*200	16	EGA
<code>_HRES16COLOR</code>	ЦГ	640*200	16	EGA
<code>_ERESNOCOLOR</code>	МГ	640*350	1	EGA
<code>_ERESCOLOR</code>	ЦГ	640*350	16	EGA
<code>_VRES2COLOR</code>	ЦГ	640*480	2	VGA
<code>_VRES16COLOR</code>	ЦГ	640*480	16	VGA
<code>_MRES256COLOR</code>	ЦГ	320*200	256	VGA

¹ М — монохромный, Ц — цветной, Т — текстовый, Г — графический

² Размер задается в строках и столбцах для текстового режима и в пикселях для графического режима

Нуль в качестве возвращаемого функцией `_setvideomode` значения указывает на то, что заданный режим не поддерживается.

После завершения программы видеомонитор должен быть возвращен в то состояние, в котором он был до входа в программу. Для этого в конце программы необходимо вызвать функцию `_setvideomode` с параметром `_DEFAULTMODE`.

Скелет графической Си-программы выглядит так:

```
#include <graph.h>    /* Требуется для графических функций */
main()
{
    if(!_setvideomode(режим)) {
        printf("Данный режим не поддерживается\n");
        exit(-1);
    }
    ...
    Тело графической программы
    ...
    _setvideomode(_DEFAULTMODE);
}
```


13.2. ДОСТУП К КОНФИГУРАЦИИ ВИДЕОРЕЖИМА

После того как установлен видеорежим, можно определить характеристики выбранного режима, используя функцию `_getvideoconfig`. Функция `_getvideoconfig` помещает в структуру предопределенного типа `videoconfig` информацию о графической конфигурации:

```
struct videoconfig {
    short numxpixels;      /* Число пикселей по горизонтали */
    short numypixels;      /* Число пикселей по вертикали */
    short numtextcols;     /* Число столбцов */
    short numtextrows;     /* Число строк */
    short numcolors;       /* Число доступных цветов */
    short bitsperpixels;   /* Число битов на пиксель */
    short numvideopages;   /* Число доступных видеостраниц */
    short mode;            /* Текущий видеорежим */
    short adapter;         /* Активный адаптер */
    short monitor;         /* Активный монитор */
    short memory;          /* Память видеомонитора
                           (в Кбайтах) */
};
```

Например, видеорежим `_ERESCOLOR` для адаптера EGA имеет следующую конфигурацию:

Пикселей по горизонтали	640
Пикселей по вертикали	350
Число столбцов	80
Число строк	43 или 25
Количество цветов	16
Битов на пиксель	4
Видеостраниц	2
Номер видеорежима	16
Тип адаптера	4 (EGA)
Тип видеомонитора	Зависит от установки
Память адаптера	256

Доступ к конфигурации видеорежима с использованием функции `_getvideoconfig` показан в примере 1 программы, которая распечатывает конфигурацию видеорежимов.

Пример 1.

```
#include <stdio.h>
#include <graph.h>
/* Массив видеорежимов, поддерживаемых графической
 * библиотекой Си
 */
short modes[] = {
    _TEXTBW40,      _TEXTC40,      _TEXTBW80,
    _TEXTC80,      _MRES4COLOR,    _MRESNOCOLOR,
    _HRESBW,       _TEXTMONO,
    _MRES16COLOR,   _HRES16COLOR,   _ERESNOCOLOR,
    _ERESCOLOR,    _VRES2COLOR,    _VRES16COLOR,
    _MRES256COLOR, _DEFAULTMODE
};
```



```

    };
    /* Массив имен видеорежимов */
    char *names[] = {
        "TEXTBW40",      "TEXTC40",      "TEXTBW80",
        "TEXTC80",      "MRES4COLOR", "MRESNOCOLOR",
        "HRESBW",      "TEXTMONO",
        "MRES16COLOR", "HRES16COLOR", "ERESNOCOLOR",
        "ERESCOLOR",   "VRES2COLOR", "VRES16COLOR",
        "MRES256COLOR", "DEFAULTMODE"
    };

    main()
    {
        short i,
            /* Количество элементов в массиве видеорежимов */
            num = sizeof(modes) / sizeof(modes[0]);
        /* Структура, описывающая конфигурацию видеорежима */
        struct videoconfig vc;
        /* Цикл, пока не будут обработаны все режимы */
        for(i = 0; i < num; i++) {
            /* Попытаться установить видеорежим */
            if (_setvideomode(modes[i])) {
                _getvideoconfig(&vc);      /* Видеорежим установлен */
                /* Получить его конфигурацию */
                /* Распечатать информацию о видеорежиме */
                fprintf(stdprn, "Видеорежим: %s\n", names[i]);
                fprintf(stdprn, "Пикселей по оси X: %d\n", vc.numxpixels);
                fprintf(stdprn, "Пикселей по оси Y: %d\n", vc.numypixels);
                fprintf(stdprn, "Число столбцов: %d\n", vc.numtextcols);
                fprintf(stdprn, "Число строк: %d\n", vc.numtextrows);
                fprintf(stdprn, "Количество цветов: %d\n", vc.numcolors);
                fprintf(stdprn, "Битов на пиксель: %d\n",
                    vc.bitsperpixel);
                fprintf(stdprn, "Страниц видеомонитора: %d\n",
                    vc.numvideopages);
                fprintf(stdprn, "Номер видеорежима: %d\n", vc.mode);
                fprintf(stdprn, "Адаптер: %d\n", vc.adapter);
                fprintf(stdprn, "Видеомонитор: %d\n", vc.monitor);
                fprintf(stdprn, "Память видеомонитора: %d\n\n",
                    vc.memory);
            }
            else /* Видеорежим не установлен */
                fprintf(stdprn, "Видеорежим %s не поддерживается\n\n",
                    names[i]);
        }
    }
}

```

При написании графической программы следует пользоваться не абсолютными значениями, задающими характеристики экрана видеомонитора, а данными из структуры, описывающей конфигурацию видеорежима. Это обеспечит переносимость программы на другие конфигурации.

Кроме функции `_setvideomode`, имеется функция `_setvideomoderows`, которая также предназначена для установки видеорежима, но она позволяет установить число строк экрана, используемых при работе в текстовом режиме.

Пример 2 представляет функцию, которая устанавливает видеорежим с оптимальным сочетанием разрешающей способности и цветовой гаммы.

Пример 2.

```
#include <graph.h>
/* Массив видеорежимов */
short modes[] = {
    _VRES16COLOR, _MRES256COLOR, _VRES2COLOR, /* VGA */
    _ERESCOLOR, _HRES16COLOR, _MRES16COLOR, /* EGA */
    _ERESNOCOLOR,
    _TEXTMONO, /* MA */
    _MRES4COLOR, _HRESBW, _MRESNOCOLOR, /* CGA */
    _TEXTC80, _TEXTBW80, _TEXTC40, _TEXTBW40
};
/* Массив допустимых чисел строк экрана */
short rows[] = {
    60, 50, 43, 30, 25
};
short set_vmode(void)
{
    short i, j,
    /* Размер массивов modes и rows */
    nm = sizeof(modes) / sizeof(modes[0]),
    nr = sizeof(rows) / sizeof(short);
    for (i = 0; i < nm; i++)
        for (j = 0; j < nr; j++)
            if (_setvideomoderows(modes[i], rows[j]))
                return(modes[i]);
    return(-1);
}
```

13.3. СИСТЕМА КООРДИНАТ

Понятие системы координат используется для того, чтобы обозначить положение пикселя относительно горизонтальной и вертикальной осей. В графическом режиме каждый пиксель на экране адресуется посредством пары координат.

Библиотека графических функций поддерживает три системы координат: абсолютную, относительную и масштабированную.

В *абсолютной системе координат* начало координат находится в верхнем левом углу экрана. В этой точке обе координаты имеют нулевое значение. Значение координаты *x* возрастает слева направо, а значение координаты *y* — сверху вниз. В абсолютной системе координат

нат значения координат могут принимать только положительные значения в диапазоне:

- от нуля до числа пикселей по оси x для координаты x ;
- от нуля до числа пикселей по оси y для координаты y .

Число пикселей по осям x и y зависит от конкретного графического режима видеомонитора. Эти значения доступны через поля `numxpixels` и `numypixels` структуры типа `videoconfig`, возвращаемой функцией `_getvideoconfig`.

Относительная система координат создается перемещением начала координат в любую позицию на экране. Для этого используется функция `_setvieworg`:

```
_setvieworg (short x, short y);
```

Функция `_setvieworg` перемещает начало координат экрана в точку, абсолютные координаты которой (x, y) .

Первоначально, до выполнения функции `_setvieworg`, относительная система координат совпадает с абсолютной. Для переустановки начала координат в центр экрана в любом графическом режиме видеомонитора можно использовать, например, следующие операторы:

```
_getvideoconfig(&vc);  
_setvieworg(vc.numxpixels/2-1, vc.numypixels/2-1);
```

По умолчанию активной областью для вывода графических функций является вся область экрана. При этом имеется возможность в качестве активной области вывода графических функций определить часть экрана. Для этого в библиотеке имеется две функции `_setcliprgn` и `_setviewport`:

```
_setcliprgn (short x1, short y1, short x2, short y2);  
_setviewport (short x1, short y1, short x2, short y2);
```

Абсолютные координаты $(x1, y1)$ и $(x2, y2)$ определяют соответственно верхний левый и нижний правый углы прямоугольника, ограничивающего активную область вывода.

Область, определенная с помощью функций `_setcliprgn` и `_setviewport`, называется *портом экрана*. При этом функция `_setcliprgn` не изменяет относительную систему координат в порте экрана, а функция `_setviewport` позволяет ее изменить установкой начала относительных координат в верхний левый угол порта.

В активной области экрана (будь то весь экран или порт экрана) можно установить *масштабированные координаты* с помощью функции `_setwindow`.

Функция `_setwindow` позволяет определить, во-первых, новые минимальные и максимальные значения координат x и y и, во-вторых, направление осей координат. Синтаксическая модель этой функции:


```
_setwindow (coord, wx1, wy1, wx2, wy2);
short coord;      /* Направление осей масштабированных */
                  /* координат */
double wx1, wy1;  /* Новые значения координат верх- */
                  /* него левого угла — минимальные */
                  /* значения координат x и y */
double wx2, wy2;  /* Новые значения координат нижне- */
                  /* го правого угла — максимальные */
                  /* значения координат x и y */
```

Аргумент `coord` определяет направление осей масштабированных координат и может принимать значение одной из predefined констант:

`_SCRN_COORD` — значение координаты `y` возрастает сверху вниз (обычные координаты экрана);

`_CRTS_COORD` — значение координаты `y` возрастает снизу вверх (декартовы координаты).

Для каждой точки внутри активной области, в которой установлены масштабированные координаты, в любой момент времени могут быть получены значения ее координат в любой из трех систем. Для этой цели в библиотеке Си имеется группа функций, которые получают соответствующие координаты:

`_getphyscoord` — абсолютные;

`_getviewcoord`, `_getviewcoord_w` — относительные;

`_getwindowcoord` — масштабированные.

Функция `_getphyscoord` преобразует относительные координаты точки (x, y) в абсолютные координаты и возвращает их в структуре типа `xycoord`.

Функция `_getviewcoord` преобразует абсолютные координаты точки (x, y) в относительные координаты, а `_getviewcoord_w` — масштабированные координаты точки (wx, wy) в относительные координаты.

Функция `_getwindowcoord` преобразует относительные координаты точки (x, y) в масштабированные координаты.

Если в какой-то момент необходимо определить координаты текущей точки на экране, надо воспользоваться функциями `_getcurrentposition` или `_getcurrentposition_w`. Первая из них возвращает относительные координаты текущей точки в структуре типа `xycoord`, а вторая — масштабированные координаты в структуре типа `wxycoord`. Обе структуры определены в файле `graph.h`.

Определение структуры `xycoord` (абсолютные или относительные координаты точки):

```
struct xycoord {
    short xcoord;
    short ycoord;
};
```


Определение структуры `_wxcoord` (масштабированные координаты точки):

```
struct _wxcoord {
    double wx;
    double wy;
};
```

Примечание. Все функции графической библиотеки, имена которых оканчиваются парой литер `"_w"`, работают с масштабированными координатами.

13.4. УПРАВЛЕНИЕ ЦВЕТОМ

Из табл. 13.1 видно, что для всех цветных типов адаптеров (CGA, EGA и VGA) используются два цветных текстовых режима и целая группа цветных графических режимов.

Цветные текстовые режимы (`_TEXTSC40` и `_TEXTSC80`) позволяют задавать 16 цветов для текста и 8 цветов для фона. Обычно используется режим `_TEXTSC80`.

В текстовом режиме для каждого символа, выдаваемого на экран, требуется два байта памяти видеодисплея. Первый байт (расположенный по четному адресу), содержит код ПЭВМ символа, а второй байт (расположенный по нечетному адресу) содержит атрибут символа (или атрибут изображения символа). Атрибут символа определяет особенности отображения символа на экране. Он задает цвет символа, цвет фона, а также определяет, как символ отображается на экране — с мерцанием или без.

Структура байта атрибута символа приведена на рис. 26:

Младшие 4 бита (биты 4-7) определяют цвет символа:

биты 5-7 — цвет символа;

бит 4 — интенсивность цвета.

Старшие 4 бита (биты 0-3) определяют цвет фона:

биты 1-3 — цвет фона;

бит 0 — признак мерцания.



Рис. 26. Структура байта атрибута символа

Поскольку цветной дисплей имеет три основных цвета (красный, зеленый и синий), то для определения цвета отводятся три бита.

Значения битов 1-3 и 5-7:

- 000 — черный цвет
- 001 — синий цвет
- 010 — зеленый цвет
- 011 — голубой цвет
- 100 — красный цвет
- 101 — фиолетовый цвет
- 110 — коричневый цвет
- 111 — белый цвет

Включение бита интенсивности для цвета символа определяет еще 8 цветов для символа.

Управление цветом в текстовом режиме осуществляют следующие четыре функции:

- `_gettextcolor` — получает текущий цвет текста;
- `_setttextcolor` — устанавливает цвет текста;
- `_getbkcolor` — получает текущий цвет фона;
- `_setbkcolor` — устанавливает цвет фона.

Синтаксические модели этих функций:

```
long _getbkcolor (void);
long _setbkcolor (bkcolor);
short _gettextcolor (void);
short _setttextcolor (color);
long bkcolor;           /* Цвет фона */
short color;            /* Цвет текста */
```

Функции `_setttextcolor` и `_setbkcolor` возвращают значение предыдущего цвета текста и фона.

Значения аргумента цвета для функции `_setttextcolor` приведены в табл. 13.2.

Таблица 13.2

Значения аргумента цвета функции `_setttextcolor`

Значе- ние	Цвет	Значение	Цвет
0	Черный	8	Темно-серый
1	Синий	9	Светло-синий
2	Зеленый	10	Светло-зеленый
3	Голубой	11	Светло-голубой
4	Красный	12	Светло-красный
5	Фиолетовый	13	Светло-фиолетовый
6	Коричневый	14	Желтый
7	Светло-серый	15	Белый

Кроме того, значения цвета от 16 до 31 — это те же цвета, что и от 0 до 15, но только с мерцанием.

Функции установки цвета не влияют на вывод стандартными функциями Си, например, функцией `printf`.

Для вывода цветного текста используется функция `_outtext`:

```
void _outtext (text);
char * text;      /* Выводимый текст */
```

Текст выводится, начиная с текущей текстовой позиции. Для установки текущей текстовой позиции в нужную точку экрана используется функция `_settextposition`:

```
_settextposition (row, col);
short row, col;    /* Новые координаты текущей текстовой
                    позиции */
```

Функция `_settextposition` может быть использована и со стандартными функциями вывода на экран.

Следующий пример 1 представляет программу для установки любого сочетания цвета текста и цвета фона на цветном дисплее. На экран выводятся два столбца цветowych прямоугольников, по шестнадцать прямоугольников в каждом. Левый столбец содержит допустимые цвета текста, а правый столбец — цвета фона: восемь основных цветов фона и восемь цветов с мерцанием. Выбор нужного цвета осуществляется клавишами управления курсором ↑ и ↓ путем перемещения стрелки выбора. Сначала выбирается цвет текста, а затем цвет фона. Переход от выбора цвета текста к выбору цвета фона осуществляется клавишей табуляции. Нажатие клавиши Вых после выбора цвета фона фиксирует выбранное сочетание цветов. Программа используется с адаптерами CGA, EGA и VGA.

Пример 1.

```
#include <graph.h>
#include <conio.h>
#include <stdio.h>
/* Клавиши управления выбором */
#define ESC 0x1b /* Для выхода из программы */
#define TAB 0x09 /* Для перехода от текста к фону */
#define UP 0x48 /* Для выбора предыдущего цвета */
#define DOWN 0x50 /* Для выбора последующего цвета */
main()
{
    register i, j;
    int key; /* Код нажатой клавиши */
    short tcolor = 0; /* Выбранный цвет текста */
    long bcolor = 0; /* Выбранный цвет фона */
    short row,
        col;
    /* Прототип функции, генерирующей сигнал */
    void bell(void);
```



```

    /* Установить текстовый режим с 16 цветами */
if (!_setvideomode(_TEXT80))
    return (-1);
_settextposition(4, 35);    /* Вывести заголовки */
_outtext("Цвет");
_settextposition(4, 42);
_outtext("Фон");
    /* Вывести все возможные цвета текста */
for (j = 0, row = 6, col = 36; j < 16; j++, row++) {
    _settextcolor(j);
    _settextposition(row, col);
    _outtext("\x0db\x0db\x0db");
}
    /* Вывести все возможные цвета фона */
for (j = 0, row = 6, col = 42; j < 16; j++, row++) {
    if (j < 8)    /* Восемь возможных цветов фона и */
        _settextcolor(j); /* восемь цветов с мерцанием */
    else
        _settextcolor(j + 16);
    _settextposition(row, col);
    _outtext("\x0db\x0db\x0db");
}
    /* Установить первоначально стрелку выбора */
    /* на первый цвет столбца цветов текста */
    _settextcolor(7);
    _settextposition(6, 35);
    _outtext("\x10");
    /* Цикл выбора цвета текста */
for (i = 0; ;) {
    key = getch();    /* Ожидание нажатия клавиши */
    /* Нажата клавиша TAB — сделан выбор */
    /* цвета текста. Выйти из цикла */
    if (key == TAB)
        break;
    if (key == 0) {
        /* Нажата клавиша перемещения курсора — второй раз
        * вызвать функцию getch
        */
        key = getch();
        j = i;
        switch (key) {
            /* Нажата клавиша ↑ — переместить стрелку выбора */
            /* вверх на один цвет. Если выход за пределы */
            /* цветов, выдать звуковой сигнал */
            case UP: if (i == 0)
                bell();
                else
                    i--;
                break;
            /* Нажата клавиша ↓ — переместить стрелку выбора */
            /* вниз на один цвет. Если выход за пределы */
            /* цветов, выдать звуковой сигнал */
            case DOWN: if (i == 15)

```



```

        bell();
    else
        i++;
    break;
}
}

/* Переместить стрелку выбора */
_settextposition(j + 6, 35);
_outtext(" ");
_settextposition(i + 6, 35);
_outtext("\x10");
}
tcolor = i; /* Сохранить выбранный цвет текста */
_settextposition(6, 41); /* Установить стрелку выбора */
_outtext("\x10"); /* на первый цвет столбца */
/* цветов фона */
/* Цикл выбора цвета фона */
for (i = 0; ; ) {
    key = getch();
    if (key == ESC) /* Нажата клавиша Выход — */
        break; /* сделан выбор цвета фона */
    /* Выйти из цикла */
    if (key == 0) {
        /* Нажата клавиша перемещения курсора — второй раз */
        /* вызвать функцию getch */
        key = getch();
        j = i;
        switch (key) {
            /* Нажата клавиша ↑ — переместить стрелку выбора */
            /* вверх на один цвет. Если выход за пределы */
            /* цветов, выдать звуковой сигнал */
            case UP: if (i == 0)
                bell();
                else
                    i--;
                break;
            /* Нажата клавиша ↓ — переместить стрелку выбора */
            /* вниз на один цвет. Если выход за пределы */
            /* цветов, выдать звуковой сигнал */
            case DOWN: if (i == 15)
                bell();
                else
                    i++;
                break;
        }
    }
}

/* Переместить стрелку выбора */
_settextposition(j + 6, 41);
_outtext(" ");
_settextposition(i + 6, 41);
_outtext("\x10");
}
bcolor = i; /* Сохранить выбранный цвет фона */

```



```

if (bcolor < 8) {      /* Установить выбранные цвета */
    /*   текста и фона   */
    _settextcolor(tcolor);
    _setbkcolor(bcolor);
}
else {
    _settextcolor(tcolor + 16);
    _setbkcolor(bcolor - 8);
}
_clearscreen(_GCLEARSCREEN);    /* Очистить экран */
return (0);
}

/* Функция выдачи звукового сигнала */
void bell(void)
{
    putchar('\a');
}

```

В графическом режиме для представления пикселя может быть использовано разное количество видеопамати в зависимости от типа адаптера и выбранного видеорежима. Пиксель может быть представлен одним, двумя, четырьмя или восемью битами. Количество битов для представления пикселя можно получить из поля `bitsperpixels` структуры типа `videoconfig` после установки видеорежима и вызова функции `_getvideoconfig`. Значения, которые могут быть представлены в этих битах, называются значениями пикселя. Значение пикселя определяет цвет, которым пиксель высвечивается на экране дисплея. Следовательно, количество битов, используемых для представления пикселя, определяет количество цветов, которые может иметь пиксель. Однобитовое представление определяет два цвета, двухбитовое — четыре и т.д.

Каждый цвет определяется уникальным значением или номером в множестве значений пикселя. Такое нумерованное представление цветов, характерное для каждого графического видеорежима, называется палитрой цветов. Палитра может быть неизменяемой, т.е. ее нельзя переопределить в программе (палитра видеорежимов адаптера CGA), или изменяемой, которую программист может программно переопределить (палитры видеорежимов EGA и VGA).

Рассмотрим графические видеорежимы, которые подразделяются на видеорежимы, поддерживаемые адаптерами CGA, EGA и VGA.

Для адаптера CGA предназначены видеорежимы `_MRES4COLOR`, `_MRESNOCOLOR` и `_HRESBW`. Первые два видеорежима осуществляют вывод в четырех цветах, а последний — только вывод в черно-белом изображении. Режимы `_MRES4COLOR` и `_MRESNOCOLOR` используют одну из предопределенных палитр. Палитра адаптера CGA представляет собой подмножество всех возможных цветов (всего 16 цветов) и состоит из цвета фона и трех цветов изображения. Обеспе-

чиваются четыре палитры для видеорежима `_MRES4COLOR` и две палитры для видеорежима `_MRESNOCOLOR`. Номера палитр и соответствующие им цвета для видеорежимов `_MRES4COLOR` и `_MRESNOCOLOR` приведены в табл. 13.3.

Таблица 13.3

Палитры режимов адаптера CGA

Номер палитры	Значение пикселя		
	1	2	3
Видеорежим <code>_MRES4COLOR</code>			
0	Зеленый	Красный	Коричневый
1	Голубой	Фиолетовый	Светло-серый
2	Светло-зеленый	Светло-красный	Желтый
3	Светло-голубой	Светло-фиолетовый	Белый
Видеорежим <code>_MRESNOCOLOR</code>			
0	Синий	Красный	Светло-серый
1	Светло-синий	Светло-красный	Белый

Для выбора одной из predetermined палитр предназначена функция `_selectpalette`:

```
short _selectpalette (palette);
short palette; /* Номер палитры */
```

С адаптером EGA используются видеорежимы `_MRES16COLOR`, `_HRES16COLOR` и `_ERESCOLOR`, которые обеспечивают цветную графику в шестнадцати допустимых для адаптера цветах. Палитра, используемая с адаптером EGA, состоит из 16 цветов и, в отличие от адаптера CGA, может быть переопределена в программе. Палитра, используемая по умолчанию, совпадает с приведенной в табл. 13.2.

Видеорежимы адаптера CGA можно использовать и с адаптером EGA, но только с соответствующей этим режимам цветовой гаммой.

Цветную графику с адаптером VGA обеспечивают видеорежимы `_VRES2COLOR`, `_VRES16COLOR` и `_MRES256COLOR`. Аппаратура VGA поддерживает палитру из 262 144 цветов.

Видеорежим `_VRES2COLOR` обеспечивает вывод в двух цветах, видеорежим `_VRES16COLOR` — в 16 цветах, а `_MRES256COLOR` — в 256 из допустимых в VGA цветах.

Шестнадцать цветов палитры по умолчанию для видеорежима `_VRES16COLOR` и первые шестнадцать цветов палитры по умолча-

нию видеорежима `_MRES256COLOR` совпадают с цветами, перечисленными в табл. 13.2.

Палитры видеорежимов адаптера VGA также могут быть переопределены.

Видеорежимы для адаптеров CGA и EGA можно использовать с адаптером VGA.

Для переопределения палитр видеорежимов адаптеров EGA и VGA в Си-программе предназначены функции `_remappalette` и `_remappalette`.

Функция `_remappalette` переопределяет отображение только для одного значения пикселя:

```
long _remappalette (pixel, color);
short pixel;        /* Переопределяемое значение пикселя */
long color;         /* Новый цвет, назначаемый пикселю,
                     представленному первым параметром */
```

Цвет задается в виде одной из следующих предопределенных констант:

<code>_BLACK</code> (черный)	<code>_GRAY</code> (темно-серый)
<code>_BLUE</code> (синий)	<code>_LIGHTBLUE</code> (светло-синий)
<code>_GREEN</code> (зеленый)	<code>_LIGHTGREEN</code> (светло-зеленый)
<code>_CYAN</code> (голубой)	<code>_LIGHTCYAN</code> (светло-голубой)
<code>_RED</code> (красный)	<code>_LIGHTRED</code> (светло-красный)
<code>_MAGENTA</code> (фиолетовый)	<code>_LIGHTMAGENTA</code> (светло-фиолетовый)
<code>_BROWN</code> (коричневый)	<code>_LIGHTYELLOW</code> (желтый)
<code>_WHITE</code> (светло-серый)	<code>_BRIGHTWHITE</code> (белый)

Например, после вызова функции

```
_remappalette (2, _LIGHTYELLOW);
```

все, что выдавалось в зеленом цвете (индекс зеленого цвета в палитре по умолчанию равен 2), будет выдаваться желтым цветом.

Функция `_remappalette` переопределяет отображение одновременно всех значений пикселя:

```
short _remappalette (remap_colors);
long remap_colors[]; /* Массив цветов, отражающий
                     изменение отображения */
```

Переопределение выполняется следующим образом: первый цвет в массиве будет новым цветом нулевого значения пикселя, второй цвет будет новым цветом, соответствующим значению пикселя, равному 1 и т.д. Число отображаемых цветов в палитре зависит от числа цветов в палитре, поддерживаемой текущим видеорежимом.

Установка текущего цвета для любого графического видеорежима обеспечивается функцией `_setcolor`:

```
_setcolor (fgcolor);
short fgcolor;        /* Цвет */
```


Параметр функции `_setcolor` является индексом в палитре цветов. Нулевое значение означает, что выбран цвет, совпадающий с цветом фона. Цветом по умолчанию является цвет с наибольшим индексом в текущей палитре.

Для установки фона используется функция `_setbkcolor`, параметром для которой является одна из перечисленных выше констант цвета.

Установка цвета и смена палитр показаны в примере 2 программы, наносящей точки на экран дисплея. Случайным образом определяются координаты точки. Если точка уже есть на экране, она удаляется с экрана. В противном случае точка наносится на экран. Цвет точки также определяется случайным образом. Для нанесения точек используются функции `_getpixel` и `_setpixel`, описанные в следующем пункте. Через каждые 15 секунд меняется палитра цветов. Для смены палитры используются функции `_selectpalette`, если программа выполняется с адаптером CGA, или `_remapallpalette`, если программа выполняется с адаптером EGA или VGA. Для определения типа адаптера используются предопределенные константы `_CGA`, `_EGA` и `_VGA`, определенные в файле `graph.h`.

Пример 2.

```
#include <graph.h>
#include <conio.h>
#include <stdio.h>
#include <time.h>
/* Массив цветов */
long colors[] = {
    _BLUE, _GREEN, _CYAN, _RED, _MAGENTA,
    _BROWN, _WHITE, _GRAY, _LIGHTBLUE, _LIGHTGREEN,
    _LIGHTCYAN, _LIGHTRED, _LIGHTMAGENTA,
    _LIGHTYELLOW, _BRIGHTWHITE
};
/* Массив отображаемых цветов */
long remap_colors[16];
struct videoconfig vc; /* Видеоконфигурация */
int main()
{
    short i,
        adapter, /* Адаптер */
        mode,    /* Видеорежим */
        num,     /* Количество цветов в палитре */
        x, y,    /* Координаты пикселя */
        cur_color; /* Текущий цвет */
    short set_vmode(); /* Прототип функции установки
                       видеорежима */
    /* Установить видеорежим с наилучшим сочетанием
     * разрешающей способности и цветовой гаммы
     */
    if ((mode = set_vmode()) < _MRES4COLOR ||
        mode == _TEXTMONO) {
        _setvideomode(_DEFAULTMODE);
```



```

    return (-1);
}
/* Получить видеоконфигурацию */
_getvideconfig(&vc);
/* Получить тип адаптера */
adapter = vc.adapter;
/* В зависимости от типа адаптера определить количество
 * цветов в палитре
 */
switch (adapter) {
    case _CGA: num = 3;
        break;
    case _EGA:
    case _VGA: num = 16;
        break;
    default: return (-1);
}
/* Цикл, пока не будет нажата клавиша */
while (!kbhit()) {
    x = rand() % vc.numxpixels; /* Получить координаты */
    y = rand() % vc.numypixels; /* пикселя */
    if (_getpixel(x, y) != 0) /* Зажечь или погасить */
        cur_color = 0; /* пиксель */
    else
        cur_color = rand() % (num - 1) + 1;
    _setcolor(cur_color);
    _setpixel(x, y);
    /* Через каждые 15 секунд менять палитру цветов, не */
    /* меняя фон */
    if (time(NULL) % 15 == 0)
        switch (adapter) {
            case _CGA: _selectpalette(rand() % 4);
                break;
            case _EGA:
            case _VGA: for (i = 1; i < 16; i++)
                    remap_colors[i] = colors[rand() % 15];
                    _remapallpalette(remap_colors);
                    break;
        }
}
/* Установить исходный видеорежим и выйти из */
/* программы */
_setvideomode(_DEFAULTMODE);
return (0);
}

```

13.5. НАНЕСЕНИЕ ТОЧЕК И ВЫЧЕРЧИВАНИЕ ЛИНИЙ

Простейшие базовые графические функции оперируют с одним пикселем. С пикселем можно выполнять следующие операции:

- проанализировать пиксель, т.е. проверить, установлен ли он на какой-либо цвет;

- установить пиксель на определенный цвет. Точка наносится на экран установкой пикселя на текущий цвет;
- погасить пиксель.

Функции анализа пикселя:

```
short _getpixel (x, y);
short _getpixel_w (wx, wy);
short x, y;      /* Относительные координаты пикселя */
short wx, wy;    /* Масштабированные координаты */
                 /* пикселя */
```

Функции анализа пикселя возвращают значение (цвет) пикселя.

Функции установки пикселя на текущий цвет:

```
short _setpixel (x, y);
short _setpixel_w (wx, wy);
short x, y;      /* Относительные координаты пикселя */
short wx, wy;    /* Масштабированные координаты */
                 /* пикселя */
```

Гашение пикселя выполняется установкой его на цвет фона, который всегда соответствует нулевому значению. Последовательность команд для гашения пикселя должна быть следующей:

```
_setcolor(0);    /* Установить текущий цвет на цвет фона */
_setpixel(x, y); /* Погасить пиксель в точке (x,y) */
```

Для нанесения последовательности точек на экран функция `_setpixel` (`_setpixel_w`) используется в совокупности с функцией `_moveto` (`_moveto_w`), которая перемещает курсор в нужную точку на экране.

Прототипы функций `_moveto` и `_moveto_w`:

```
_moveto (short x, short y);
_moveto_w (double wx, double wy);
```

Программа в примере 1 демонстрирует использование графических функций библиотеки Си для рисования на экране. Рисование выполняется перемещением рисующей точки ("карандаша"), которая первоначально появляется в центре экрана. Для перемещения карандаша используются клавиши управления перемещением курсора. Длинная клавиша пробела позволяет отключить цвет рисования, например, чтобы переместить карандаш в другое место экрана. Повторное нажатие клавиши пробела вновь включает цвет рисования.

Пример 1.

```
#include <conio.h>      /* Требуется для функции getch */
#include <graph.h>

#define ESC    0x1b     /* Клавиша выхода из программы */
#define SPACE  0x20     /* Клавиша переключения цвета */
                     /* рисования */

/* Клавиши для рисования */
```



```

#define UP      0x48 /* Рисовать вверх */
#define DOWN    0x50 /* Рисовать вниз */
#define LEFT    0x4b /* Рисовать влево */
#define RIGHT   0x4d /* Рисовать вправо */
#define HOME    0x47 /* Рисовать вправо вверх */
#define END     0x4f /* Рисовать вправо вниз */
#define PGUP    0x49 /* Рисовать влево вверх */
#define PGDOWN  0x51 /* Рисовать влево вниз */
main()
{
    struct xycoord new, /* Координаты карандаша и */
        old, /* рисуемой точки */
        phys; /* Абсолютные координаты экрана */
    /* Структура, описывающая видеоконфигурацию */
    struct videoconfig vc;
    int key; /* Код клавиши */
    short pen_color, /* Цвет карандаша */
        draw_color, /* Цвет рисуемой точки */
        mode; /* Видеорежим */
    short set_vmode(void); /* Прототип функции set_vmode */
    /* Установить видеорежим с наибольшей */
    /* разрешающей способностью */
    if ((mode = set_vmode()) <= _TEXTC80 || mode == _TEXTMONO)
    {
        _setvideomode(_DEFAULTMODE);
        return (-1);
    }
    /* Получить конфигурацию видеорежима */
    _getvideoconfig(&vc);
    /* Получить цвет карандаша и */
    /* начальный цвет рисуемой точки */
    pen_color = draw_color = _getcolor();
    /* Установить начало относительной */
    /* системы координат в центр экрана */
    old = new = _setvieworg(vc.numxpixels / 2 - 1,
        vc.numypixels / 2 - 1);
    while (1) { /* Бесконечный цикл */
        /* Установить карандаш */
        _setcolor(pen_color);
        _setpixel(new.xcoord, new.ycoord);
        key = getch(); /* Прочитать код клавиши */
        switch (key) {
            /* Нажата клавиша Выход — выйти из программы */
            /* с восстановлением первоначального */
            /* режима видеомонитора */
            case ESC: return (!_setvideomode(_DEFAULTMODE));
            /* Нажата клавиша ПРОБЕЛ — переключить */
            /* цвет рисуемой точки (с текущего цвета */
            /* на цвет фона или наоборот) */
            case SPACE: if (draw_color == pen_color)
                draw_color = 0;
                else
                draw_color = pen_color;
                continue;
            /* (для клавиши перемещения курсора функция */

```



```

/* getch должна быть вызвана дважды) — */
/* изменить координаты карандаша */
case 0: key = getch();
switch (key) {
case UP: new.ycoord -= 1;
break;
case DOWN: new.ycoord += 1;
break;
case LEFT: new.xcoord -= 1;
break;
case RIGHT: new.xcoord += 1;
break;
case HOME: new.xcoord = 1;
new.ycoord = 1;
break;
case PGUP: new.xcoord += 1;
new.ycoord = 1;
break;
case END: new.xcoord = 1;
new.ycoord += 1;
break;
case PGDOWN: new.xcoord += 1;
new.ycoord += 1;
break;
}
break;
}

/* Преобразовать координаты карандаша */
/* в абсолютные координаты экрана */
phys = _getphyscoord(new.xcoord, new.ycoord);
/* Если карандаш в пределах экрана, */
/* нанести точку на экран */
if ((phys.xcoord >= 0) && (phys.xcoord < vc.numxpixels)
&& (phys.ycoord >= 0) && (phys.ycoord < vc.numypixels))
{
_setcolor(draw_color);
_setpixel(old.xcoord, old.ycoord);
old = new; /* Изменить координаты рисуемой точки */
}
else /* Иначе ничего не рисовать */
new = old;
}
}

```

Имеется возможность вычерчивать прямые линии и дуги. Для вычерчивания линий используются функции `_lineto` и `_lineto_w`:

```

short _lineto (short x, short y);
short _lineto_w (double wx, double wy);

```

Функции вычерчивают прямую линию, начиная с текущей позиции курсора, в точку, координаты которой передаются ей в качестве аргументов.

После выполнения функции текущей позицией становится точка,

координаты которой заданы аргументами функции. Линия вычерчивается текущим цветом (т.е. цветом, установленным последней функцией `_setcolor`). Линия может быть либо непрерывной, либо заданной определенным шаблоном, так называемым стилем линии. Стилль линии определяется с помощью маски, которая представляет собой 16-битовый шаблон, каждый бит которого представляет пиксель на линии. Если бит установлен в 1, то соответствующий пиксель устанавливается на текущий цвет, если в 0 — на цвет фона. Шаблон распространяется на всю линию. По умолчанию используется маска, соответствующая непрерывной линии (`0xFFFF`).

Для установки маски стиля линии предназначена функция `_setlinestyle`:

```
_setlinestyle (short mask);
```

Например, после выполнения функции

```
_setlinestyle (0xAAAA);
```

все линии, вычерчиваемые функцией `_lineto` или `_lineto_w`, будут прерывистыми, так как значению `0xAAAA` в двоичном виде соответствует значение `1010101010101010`.

Функции `_arc` и `_arc_w` вычерчивают эллиптическую дугу:

```
_arc (x1, y1, x2, y2, x3, y3, x4, y4);
```

```
_arc_w (wx1, wy1, wx2, wy2, wx3, wy3, wx4, wy4);
```

```
short x1, y1;      /* Левый верхний угол */
```

```
double wx1, wy1;   /* ограничивающего прямоугольника */
```

```
short x2, y2;      /* Правый нижний угол */
```

```
double wx2, wy2;   /* ограничивающего прямоугольника */
```

```
short x3, y3;      /* Начальный вектор */
```

```
double wx3, wy3;   /* */
```

```
short x4, y4;      /* Конечный вектор */
```

```
double wx4, wy4;   /* */
```

Эллиптическая дуга (часть контура эллипса) вычерчивается в ограничивающем прямоугольнике. Концепция ограничивающего прямоугольника в графике Си используется при рисовании фигур (обычно дугообразных) и сохранении образов фигур в памяти. При вычерчивании фигур ограничивающий прямоугольник — это условный прямоугольник на экране, определяющий область, в которой вычерчивается дугообразная фигура. Центр ограничивающего прямоугольника становится центром вычерчиваемой фигуры.

Дуга вычерчивается текущим цветом от точки пересечения ее с вектором (x_3, y_3) до точки пересечения ее с вектором (x_4, y_4) . Схема вычерчивания эллиптической дуги показана на рис. 27.

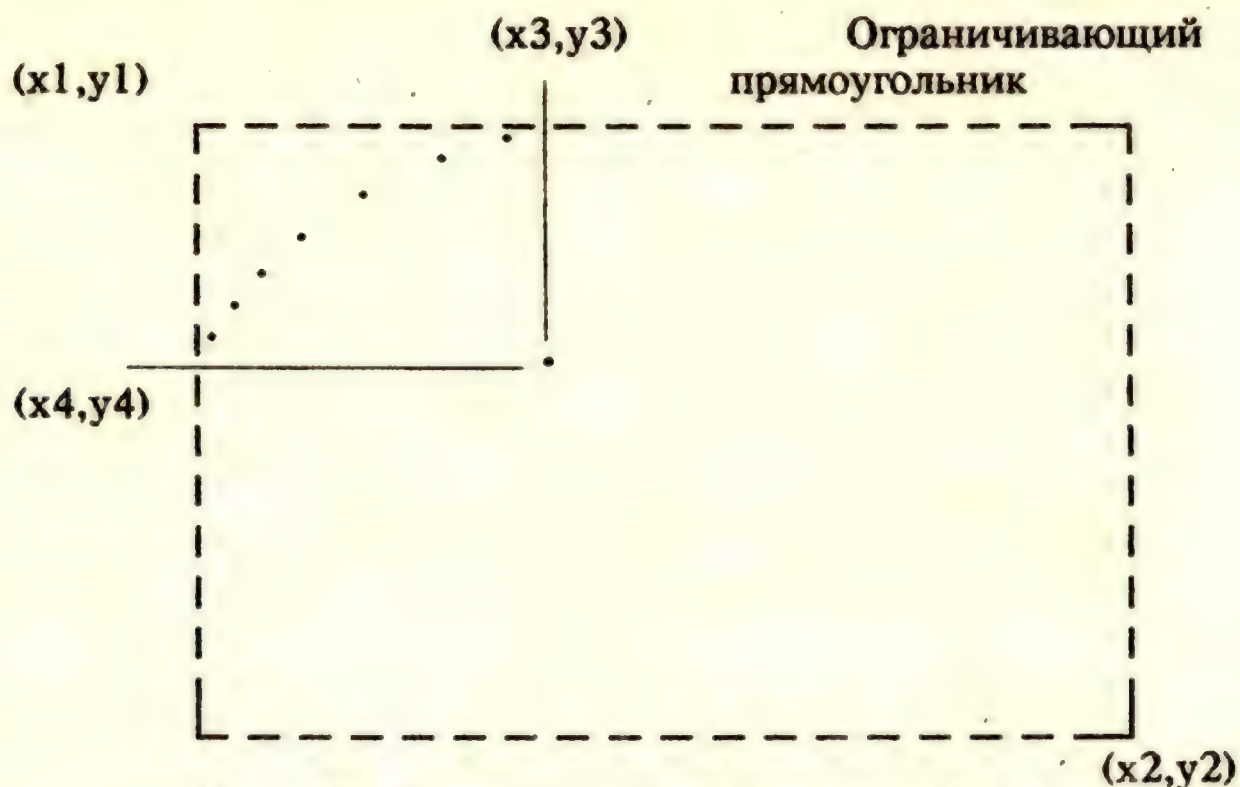


Рис. 27. Вычерчивание эллиптической дуги

Программа в примере 2 вычерчивает на экране дисплея полный период графика функции косинус от минус Пи до Пи. График вычерчивается на весь экран в масштабированных декартовых координатах. Экран масштабируется в соответствии с размерами полного периода функции косинус.

Пример 2.

```
#include <conio.h>      /* Требуется для функции getch, */
#include <math.h>        /* математических функций, */
#include <graph.h>       /* графических функций */
#define PI 3.1415926536 /* Число Пи */
main()
{
    /* Масштабированные координаты */
    struct _wxycoord wc, wcc;
    struct videoconfig vc; /* Конфигурация видеорежима */
    double xinc,          /* Приращение координаты X */
           x;             /* Координата по оси X */
    short mode;
    short set_vmode(void); /* Прототип функции set_vmode */
    /* Установить видеорежим с наибольшей разрешающей */
    /* способностью */
    if ((mode = set_vmode()) <= _TEXT80 || mode == _TEXTMONO)
    { _setvideomode(_DEFAULTMODE);
      return (-1); }
    /* Получить конфигурацию видеорежима */
    _getvideoconfig(&vc);
    /* Масштабировать экран и определить декартовы координаты */
    _setwindow(_CRTS_COORD, -PI, -1.0, PI, 1.0);
    /* Вычислить приращение координаты X */
    /* в масштабированных координатах */
    wc = _getwindowcoord(0, 0);
    wcc = _getwindowcoord(1, 1);
    xinc = wcc.wx - wc.wx;
```



```

    _moveto_w(-PI, 0); /* Нарисовать оси */
    _lineto_w(PI, 0);
    _moveto_w(0, -1.0);
    _lineto_w(0, 1.0);
    /* Вычертить график косинуса */
    for (x = -PI; x <= PI; x += xinc) {
        wc.wx = x;
        wc.wy = cos(x);
        _setpixel_w(wc.wx, wc.wy);
    }
    /* После нажатия клавиши восстановить */
    /* первоначальный режим видеомонитора и */
    /* завершить программу */
    getch(); _setvideomode(_DEFAULTMODE);
    return (0);
}

```

13.6. ВЫЧЕРЧИВАНИЕ ФИГУР

Графические примитивы библиотеки Си позволяют вычерчивать следующие базовые фигуры: прямоугольник, эллипс, эллиптический клин.

Функции `_rectangle` и `_rectangle_w` рисуют прямоугольник:

```

_rectangle (filling, x1, y1, x2, y2);
_rectangle_w (filling, wx1, wy1, wx2, wy2);
short filling;          /* Флажок закрашивания */
short x1, y1;           /* Координаты левого верхнего угла */
double wx1, wy1;
short x2, y2;           /* Координаты правого нижнего угла */
double wx2, wy2;

```

Прямоугольник задается координатами двух противоположных углов. Контур прямоугольника вычерчивается в соответствии со стилем линии, установленным функцией `_setlinestyle`.

Функции `_ellipse` и `_ellipse_w` рисуют эллипс:

```

_ellipse (filling, x1, y1, x2, y2);
_ellipse_w (filling, wx1, wy1, wx2, wy2);
short filling;          /* Флажок закрашивания */
short x1, y1;           /* Координаты левого верхнего угла */
double wx1, wy1;        /* ограничивающего прямоугольника */
short x2, y2;           /* Координаты правого нижнего угла */
double wx2, wy2;        /* ограничивающего прямоугольника */

```

Эллипс вычерчивается в ограничивающем прямоугольнике.

Функции `_pie` и `_pie_w` рисуют эллиптический клин, напоминающий по форме "кусочек пирога":

```

_pie (filling, x1, y1, x2, y2, x3, y3, x4, y4);
_pie_w (filling, wx1, wy1, wx2, wy2, wx3, wy3, wx4, wy4);
short filling;          /* Флажок закрашивания */
short x1, y1;           /* Координаты левого верхнего угла */
double wx1, wy1;        /* ограничивающего прямоугольника */

```



```

short x2, y2;      /* Координаты правого нижнего угла */
double wx2, wy2;   /* ограничивающего прямоугольника */
short x3, y3;      /* Ограничивающие векторы */
double wx3, wy3;
short x4, y4;
double wx4, wy4;

```

Эллиптический клин также вычерчивается в ограничивающем прямоугольнике. Схема вычерчивания эллиптического клина показана на рис. 28.



Рис. 28. Вычерчивание эллиптического клина

Контуры всех базовых фигур вычерчиваются текущим цветом.

Первым параметром функций, вычерчивающих базовые фигуры, является флажок закрашивания, который определяет, будет ли закрашиваться фигура. Флажок закрашивания может принимать значение одной из следующих символических констант:

_GFILLINTERIOR — вычерчиваемая фигура закрашивается текущим цветом с использованием текущей маски закрашивания;

_GBORDER — вычерчиваемая фигура не закрашивается.

Маска закрашивания представляет собой битовый шаблон размерностью 8 на 8, где каждый бит представляет пиксель. Значение "1" в бите означает, что соответствующий пиксель устанавливается на текущий цвет, значение "0" в бите оставляет пиксель без изменения. Маска распространяется на всю область закрашивания. Она строится следующим образом:

- представить маску закрашивания как массив из 8 символов;
- в сетке с 8-ю строками и 8-ю столбцами поместить единицу в каждую клетку, соответствующую высвечиваемому пикселю. В остальные клетки надо поместить нули;

- полученные в каждой строке двоичные числа преобразовать в десятичное, шестнадцатеричное или восьмеричное число без знака;
- определить массив из восьми элементов типа `unsigned char`.

Например, маска закрашивания, представляющая собой букву 'H', имеет вид:

```
unsigned char fillmask[] = {
    195, 195, 195, 255, 255, 195, 195, 195
};
```

Двоичное представление этой маски:

```
1 1 0 0 0 0 1 1
1 1 0 0 0 0 1 1
1 1 0 0 0 0 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 0 0 0 0 1 1
1 1 0 0 0 0 1 1
1 1 0 0 0 0 1 1
```

Маска, которая будет использоваться для последующего закрашивания, устанавливается функцией `_setfillmask`:

```
_setfillmask (fillmask);
unsigned char *fillmask; /* Маска закрашивания */
```

По умолчанию маска закрашивания соответствует сплошному закрашиванию (аргумент `NULL` функции `_setfillmask`).

В общем случае для закрашивания используется функция `_floodfill` (`_floodfill_w`):

```
_floodfill (x, y, bound);
_floodfill_w (wx, wy, bound);
short x, y; /* Начальная точка */
double wx, wy;
short bound; /* Цвет окантовки */
```

Функция `_floodfill` (`_floodfill_w`) позволяет закрашивать как замкнутые фигуры, так и фон. Если начальная точка (`x`, `y`) находится внутри замкнутой фигуры, то закрашивается фигура, если же начальная точка находится вне фигуры, то закрашивается фон. Закрашивание выполняется текущими цветом и маской закрашивания.

Использование функции `_floodfill` для закрашивания фигур с несплошным контуром, а также использование флажка закрашивания `_GFILLINTERIOR` в сочетании с несплошным стилем линии может привести к непредвиденным результатам.

Функции `_rectangle` (`_rectangle_w`) и `_ellipse` (`_ellipse_w`) позволяют вычерчивать квадраты и окружности. При их вычерчивании надо учитывать, что на единичном отрезке по горизонтали пикселей больше, чем на идентичном отрезке по вертикали. Поэтому необходимо ис-

пользовать коэффициент сжатия координаты Y. Для любого режима видеомонитора коэффициент сжатия вычисляется по формуле:

$$КС = (ШЭ / ВЭ) * (y / x),$$

где

КС — коэффициент сжатия,
ШЭ — физическая ширина экрана,
ВЭ — физическая высота экрана,
x, y — характеристики текущей разрешающей способности (измеряемой в пикселях).

Например, для видеомонитора типа "Электроника МС 6105", размер экрана которого 21*15 сантиметров, коэффициент сжатия может быть вычислен следующим образом:

$$КС = (21 * vc.numxpixels) / (15 * vc.numypixels);$$

Более сложные фигуры можно получить, используя комбинации описанных выше функций рисования.

Следующий пример представляет программу, которая вычерчивает восемь квадратов с учетом коэффициента сжатия и организует циклическое перемещение цветов по этим квадратам. Программа может быть использована только с адаптером EGA или VGA.

Пример.

```
#include <graph.h>
#include <conio.h>
/* Массив отображаемых цветов */
long remap_colors[] = {
    _GRAY, _BLUE, _GREEN, _CYAN,
    _RED, _MAGENTA, _BROWN, _WHITE
};
/* Массив цветов */
short colors[] = {8, 1, 2, 3, 4, 5, 6, 7};
struct videoconfig vc; /* Видеоконфигурация */
int main()
{
    short x, y,
        xinc, /* Ширина стороны квадрата */
        mode, /* Видеорежим */
        i, j, k;
    float ar; /* Коэффициент сжатия */
    short set_vmode(void); /* Прототип функции установки видеорежима */

    /* Установить видеорежим с наилучшим сочетанием
     * разрешающей способности и цветовой гаммы
     */
    if ((mode = set_vmode()) < _MRES16COLOR ||
        mode == _VRES2COLOR) {
        _setvideomode(_DEFAULTMODE);
        return (-1);
    }
    _getvideoconfig(&vc); /* Получить видеоконфигурацию */
    /* Вычислить коэффициент сжатия оси Y */
```



```

ar = (float) (10.0 * vc.numypixels) / (6.5 * vc.numxpixels);
/* Вычислить ширину стороны квадрата */
xinc = vc.numxpixels / 8;
y = xinc * ar;

/* Нарисовать восемь цветных квадратов */
for (x = 0, i = 0; i < 8; x += xinc, i++) {
    _setcolor(colors[i]);
    _rectangle(_GFillInterior, x, vc.numypixels / 2 - y / 2,
               x + xinc, vc.numypixels / 2 + y / 2);
}
k = 7; /* Указатель в массиве отображаемых цветов */
while (!kbhit()) { /* Цикл, пока не будет нажата клавиша */
    /* Цикл переотображения */
    for (i = 0, j = k; i < 8; i++) {
        _remappalette(colors[i], remap_colors[j]);
        j++;
        if (j > 7)
            j = 0;
    }
    k--; /* Изменить указатель в массиве */
    if (k < 0) /* отображаемых цветов */
        k = 7;
}
/* Установить исходный видеорежим и завершить программу */
_setvideomode(_DEFAULTMODE);
return (0);
}

```

13.7. ПЕРЕМЕЩЕНИЕ ФИГУР ПО ЭКРАНУ

Можно организовать перемещение нарисованной фигуры (изображения) по экрану. Наиболее простой способ состоит в следующем:

- нарисовать фигуру на экране;
- удалить ее с экрана;
- перерисовать фигуру на новом месте экрана и т.д.

Однако этот способ перемещения фигуры по экрану мало эффективен, поскольку прорисовка сложных фигур может занимать большой промежуток времени.

Более эффективен следующий способ:

- 1) нарисовать фигуру на экране;
- 2) сохранить изображение нарисованной фигуры в памяти;
- 3) удалить фигуру с экрана;
- 4) отобразить сохраненное в памяти изображение на новом месте экрана;
- 5) повторить с шага 3 нужное количество раз.

Для реализации этого способа перемещения объектов по экрану используются следующие функции:

`_getimage ( _getimage_w )` — сохраняет изображение в памяти;
`_putimage ( _putimage_w )` — отображает сохраненное в памяти изображение на экране.

Синтаксические модели этих функций:

```
void _getimage (x1, y1, x2, y2, image);
void _getimage_w (wx1, wy1, wx2, wy2, image);
short x1, y1,      /* Координаты ограничивающего */
      x2, y2;      /* прямоугольника */
double wx1, wy1,
      wx2, wy2;
char * image;      /* Область памяти для сохранения
                   изображения */
void _putimage (x, y, image, putting);
void _putimage_w (wx, wy, image, putting);
short x, y;        /* Координаты изображения */
double wx, wy;
char * image;      /* Область памяти, содержащая
                   изображение */
short putting;     /* Взаимосвязь с существующим */
                  /* изображением экрана */
```

Функция `_getimage` сохраняет изображение в памяти. Размер сохраняемого изображения определяется ограничивающим прямоугольником. В данном случае ограничивающий прямоугольник — это прямоугольная область на экране, охватывающая (полностью или частично) фигуру. Эта прямоугольная область экрана, содержащая фигуру, сохраняется в памяти. Область памяти, в которой сохраняется изображение, должна быть достаточного размера, чтобы вместить изображение. Для определения размера этой области предназначена функция `_imagesize ( _imagesize_w )`:

```
long _imagesize (x1, y1, x2, y2);
long _imagesize_w (wx1, wy1, wx2, wy2);
short x1, y1,      /* Координаты ограничивающего */
      x2, y2;      /* прямоугольника */
double wx1, wy1,
      wx2, wy2;
```

Функция возвращает размер памяти (в байтах), необходимой для сохранения изображения. Затем для получения необходимой памяти можно использовать одну из функций распределения памяти.

После того как изображение сохранено в памяти, оно может быть выведено в нужную точку экрана функцией `_putimage`. При вызове функции `_putimage` можно определить, каким образом будет взаимодействовать изображение в памяти с текущим изображением на экране. Этим управляет последний параметр функции, который может иметь значение одной из предопределенных констант, приведенных ниже.

Предопределенные константы функции `_putimage`

Константа	Назначение
<code>_GPSET</code>	Каждая точка передается на экран с тем атрибутом цвета, который она имела при сохранении функцией <code>_getimage</code>
<code>_GPRESET</code>	Каждая точка передается на экран с инвертированным атрибутом цвета по отношению к тому, какой она имела при получении с экрана функцией <code>_getimage</code> . Таким образом создается негативное изображение
<code>_GOR</code>	Изображение накладывается на существующее изображение, при этом предыдущее содержимое экрана не стирается
<code>_GXOR</code>	Приводит к инвертированию точек экрана, если точка существует в буфере. Если передать одно и то же изображение дважды в одну точку экрана, фон остается без изменения, а изображение стирается
<code>_GAND</code>	Передает изображение поверх существующего на экране. Результирующим изображением является результат логического умножения двух изображений

Следующий пример представляет программу, которая демонстрирует использование функций графики для перемещения изображения по экрану (шарик, бегающий по экрану).

Пример.

```
#include <graph.h>
#include <conio.h>
#include <malloc.h>
#include <stdlib.h>
#define UP    -(vc.numypixels / 2)      /* Значения,      */
#define DOWN  vc.numypixels / 2         /* определяющие  */
#define LEFT  -(vc.numxpixels / 2)     /* границы экрана */
#define RIGHT vc.numxpixels / 2
int main()
{
    struct videoconfig vc;      /* Конфигурация видеорежима */
    char * buffer;
    short x, y,                /* Радиус круга              */
           xcur,               /* Текущие координаты фигуры  */
           ycur,               /* на экране                  */
           dx, dy,             /* Приращения по осям X и Y   */
           mode;               /* Видеорежим                 */
    float ar;                  /* Коэффициент сжатия         */
    short set_vmode(void);     /* Прототип функции set_vmode */
    /* Установить видеорежим */
    if ((mode = set_vmode()) <= _TEXTC80 ||
        mode == _TEXTMONO) {
        _setvideomode(_DEFAULTMODE);
        return(-1);
    }
    /* Получить конфигурацию видеорежима */
    _getvideoconfig(&vc);
    /* Установить радиус круга с учетом коэффициента сжатия */
```



```

ar = (float)((10.0 * vc.numypixels)/(6.5 * vc.numxpixels));
x = 15;
y = 15 * ar;
_clearscreen(_GCLEARSCREEN); /* Очистить экран */
/* Установить начало логических координат в центр
экрана */
_setvieworg(vc.numxpixels / 2 - 1, vc.numypixels / 2 - 1);
_setcolor(rand() % 2 + 1); /* Установить цвет */
/* Нарисовать круг */
_ellipse(_GFILLINTERIOR, -x, -y, x, y);
/* Получить память для изображения круга */
buffer = (char *) malloc((unsigned) _imagesize(-x - 1,
-y - 1, x + 1, y + 1));
/* Сохранить изображение в памяти */
_getimage(-x - 1, -y - 1, x + 1, y + 1, buffer);
xcur = -x - 1; /* Установить начальные координаты */
ycur = -y - 1; /* изображения на экране и */
dx = 5; /* приращения по осям */
dy = -5;
while (!kbhit()) { /* Цикл, пока не будет нажата клавиша */
/* Стереть старое изображение круга */
_putimage(xcur, ycur, buffer, _GXOR);
/* Изменить координаты круга */
if (xcur + dx > RIGHT - 32) /* Если по какой-то из */
dx = -(rand() % 5 + 1); /* координат выход за */
if (xcur + dx < LEFT) /* пределы экрана, */
dx = rand() % 5 + 1; /* изменить направление */
if (ycur + dy > DOWN - 32) /* движения по этой */
dy = -(rand() % 5 + 1); /* координате на */
if (ycur + dy < UP) /* противоположное */
dy = rand() % 5 + 1;
xcur += dx;
ycur += dy;
/* Нарисовать новое изображение */
_putimage(xcur, ycur, buffer, _GXOR);
}
free(buffer); /* Освободить память */
_clearscreen(_GCLEARSCREEN); /* Очистить экран */
_setvideomode(_DEFAULTMODE); /* Восстановить исходный
видеорежим */
return(0); /* Выйти из программы */
}

```


Часть 4

ИНТЕГРИРОВАННАЯ СРЕДА РАЗРАБОТКИ СИ-ПРОГРАММ

Интегрированная среда Си является частью системы программирования Си. Основным достоинством среды Си является интеграция необходимых средств разработки Си-программ в единую среду программирования — интегрированную среду. Не выходя из среды, программист имеет возможность создавать, компилировать, выполнять, отлаживать, корректировать программу.

Глава 14

ОСНОВНЫЕ ПОНЯТИЯ СРЕДЫ

Интегрированная среда Си является удобным средством для быстрой разработки программ на языке Си. Работа пользователя в среде Си базируется на использовании техники меню и окон. К тому же в среде имеется контекстно чувствительное средство получения справочной информации. Все это делает интерфейс с компилятором легким в изучении.

Чтобы начать работу в среде Си при установленном маршруте поиска программных компонентов среды, достаточно набрать имя программы, представляющей интегрированную среду, например:

QC

14.1. ЭКРАН СРЕДЫ СИ

При выполнении команды вызова интегрированной среды появится экран, показанный на рис. 29.

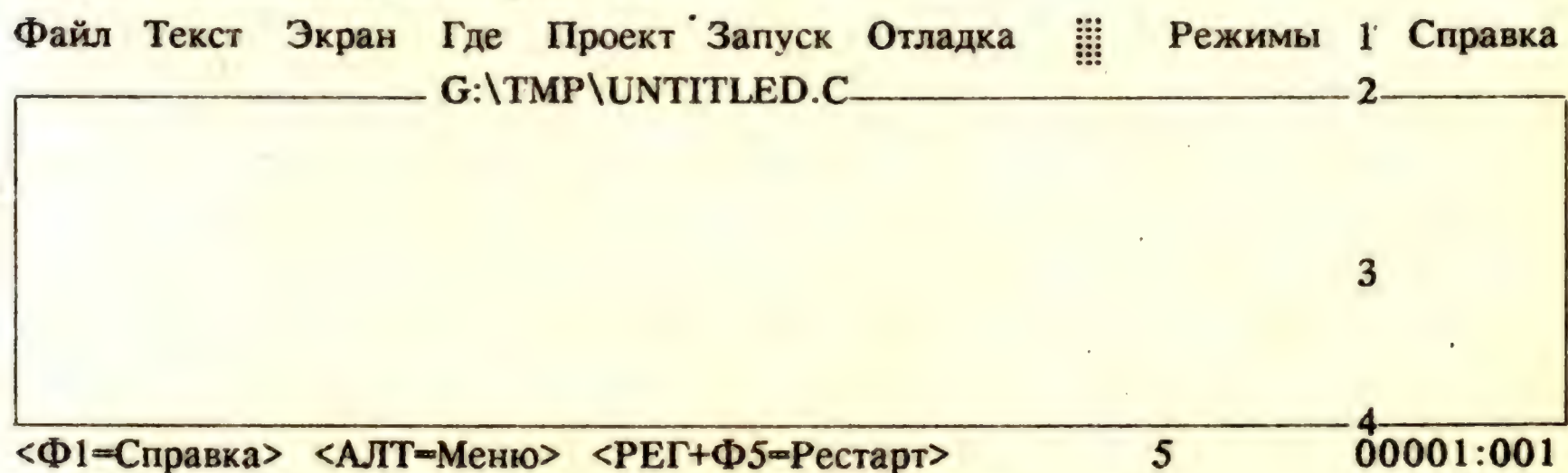


Рис. 29. Компоненты экрана среды Си:

1 — полоса меню; 2 — полоса заголовка; 3 — основное окно просмотра; 4 — полоса прокрутки; 5 — строка подсказки.

Полоса меню — полоса в верхней части экрана, содержащая имена меню среды Си.

Каждое меню из полосы меню содержит группу команд, выполняющих определенные действия. Ниже приведены виды меню среды Си и действия, которые можно выполнить с помощью команд, содержащихся в этих меню.

Меню среды Си

Имя меню	Функции команд
Файл	Создать, загрузить, объединить, печатать и сохранить исходный файл, а также временно выйти в среду Альфа-ДОС и закончить сеанс работы в среде Си
Текст	Удалить, добавить, копировать текст исходного файла
Экран	Выбрать для просмотра любой исходный файл программы, файл включения; открыть, закрыть, изменить размер любого окна среды Си, а также переключить экран среды на экран вывода программы
Где	Искать, заменить текст в исходном файле, найти функцию во время отладки, показать строки исходного файла, на которые выдаются сообщения об ошибках компиляции
Проект	Компилировать программу, строить и редактировать список исходных файлов программы (так называемый программный список), строить программу
Запуск	Выполнить программу. Выполнять программу по шагу, трассировать во время отладки
Отладка	Управлять использованием отладочных средств, таких как создание точек прерываний, наблюдение за выражениями, ведение истории отладки
Утилиты	Выполнить любую команду Альфа-ДОС, расширить стандартное меню своими командами
Режимы	Управлять цветом экрана и режимами среды Си, режимами компилятора, редактора связей, режимами времени выполнения и отладки, установкой маршрутов поиска файлов, а также установкой режима полного или короткого меню
Справка	Управлять выдачей справочной информации

Полоса заголовка — полоса, в которой содержится имя файла, редактируемого в окне просмотра исходного текста. Если файл не загружен, в полосе заголовка появляется текст "untitled.c".

Окно просмотра исходного текста является основным окном среды Си. Загруженный в это окно исходный файл может быть отредактирован, прокомпилирован, выполнен. В этом окне можно выполнять трассировку программы в режиме отладки.

Кроме этого основного окна, в среде Си есть возможность от-

крыть еще 6 окон, используемых для различных целей. Эти окна используются для просмотра информации во время отладки (окна "Отладка", "Регистры", "Локальные"), для просмотра сообщений об ошибках компиляции, редактирования, выполнения (окно "Ошибки"), для хранения и редактирования дополнительной информации (окно "Блокнот"), для получения справочной информации (окно "Справка").

Полосы прокрутки — полосы вдоль правой и нижней границ экрана. Они дают удобный способ перемещения текста в окне просмотра вверх, вниз, вправо и влево с помощью манипулятора графической информации.

Строка подсказки содержит основную справочную информацию о сокращенном наборе клавиш для выполнения возможных в данном контексте действий. Например, после входа в среду Си пользователь может, нажав клавишу F1, получить справочную информацию и, нажав клавишу ALT+F, открыть меню "Файл".

На экране среды, кроме описанных полей, высвечиваются еще два блока:

- курсор — мерцающее подчеркивание или блок, который показывает, где будет появляться на экране вводимый текст. Если среда Си находится в режиме вставки, курсор является мерцающим подчеркиванием, если в режиме замены, то курсор является мерцающим блоком;

- указатель манипулятора — маленький прямоугольный блок, который появляется в центре экрана, если ПЭВМ снабжена манипулятором графической информации и был загружен соответствующий драйвер.

14.2. ВЫБОР КОМАНД МЕНЮ

Существует несколько способов открытия меню и выбора нужной команды.

Первый способ состоит из следующей последовательности действий:

- открыть меню, нажав клавишу ALT и клавишу с первой буквой имени меню. Такая последовательность нажатий клавиш далее будет обозначаться со знаком "+" в качестве соединителя. Например, после нажатия клавиш ALT+F будет открыто меню "Файл", показанное на рис. 30;

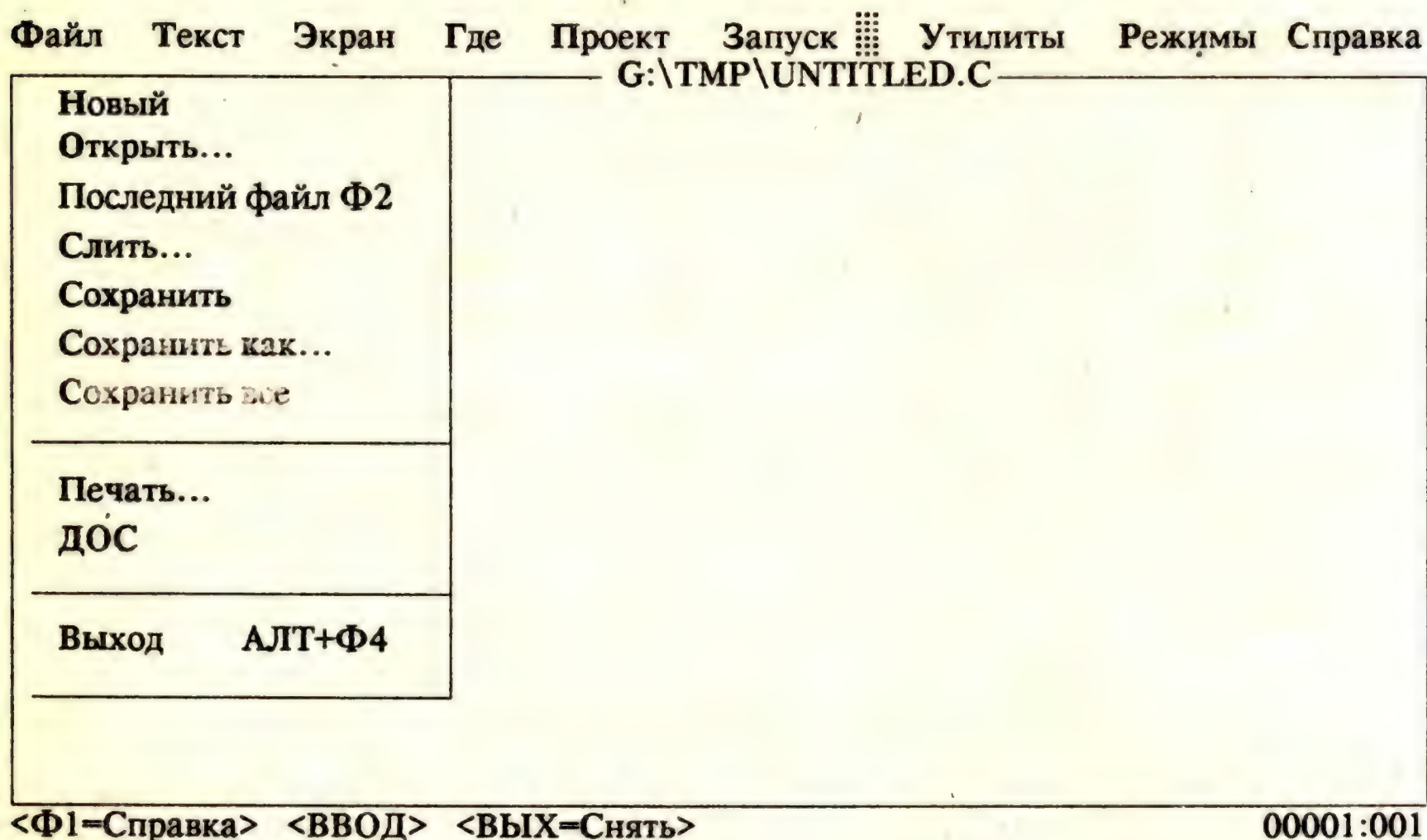


Рис. 30. Экран среды с меню "Файл"

• для выбора команды из открытого меню надо ввести ярко светящуюся или подчеркнутую букву в имени этой команды. Если же нужная команда уже высвечена, достаточно нажать клавишу ВВОД. Команды, заканчивающиеся многоточием, запрашивают у пользователя дополнительную информацию.

Если в имени команды отсутствует ярко светящаяся или подчеркнутая буква, эта команда в данной ситуации неприменима, как, например, команда "Последний файл" на рис. 30. Если сеанс только начинается, в нем не может быть последнего редактировавшегося файла.

Второй способ выбора меню и команд меню состоит в следующем:

- нажать клавишу ALT;
- перемещать светящуюся строку по именам меню в полосе меню с помощью клавиш управления курсором;
- нажать клавишу ВВОД, чтобы открыть меню, имя которого высвечено;
- в открытом меню команду можно выбирать либо по высвеченной букве, либо продвигая светящуюся строку к нужной команде с помощью клавиш управления курсором.

Чтобы аннулировать любую команду до ее выполнения, нужно нажать клавишу ВЫХ. Этой клавишей снимается любое открытое меню.

Третий способ выбора команд предоставляет манипулятор графической информации "мышь". При описании применения манипулятора будут использоваться следующие термины:

- "подвести" — перемещать манипулятор до тех пор, пока указатель не будет подведен к тому объекту, на который нужно указать;
- "нажать" — задержать нажатой вниз кнопку манипулятора;
- "щелкнуть" — быстро нажать и отпустить кнопку манипулятора. Щелкание относится к левой кнопке, если не оговорено иное;
- "протянуть" — держать в нажатом состоянии левую кнопку манипулятора при перемещении манипулятора по плоской поверхности, например по крышке стола;
- "дважды щелкнуть" — дважды последовательно щелкнуть левой кнопкой манипулятора в быстром темпе.

Для того чтобы выбрать команду с использованием манипулятора графической информации, следует щелкнуть левой кнопкой манипулятора сначала для имени меню, а затем для нужной команды. Чтобы аннулировать команду, достаточно щелкнуть где-либо вне меню левой кнопкой манипулятора.

Некоторые команды можно выполнить, не открывая меню, а просто нажав комбинацию клавиш, указанную в правой части строки команды в меню. Такие комбинации клавиш носят название "сокращенных наборов клавиш" или "быстрых ключей" и перечислены в табл. 14.1. Команды, для которых нет соответствующих комбинаций клавиш, нельзя выполнить этим способом.

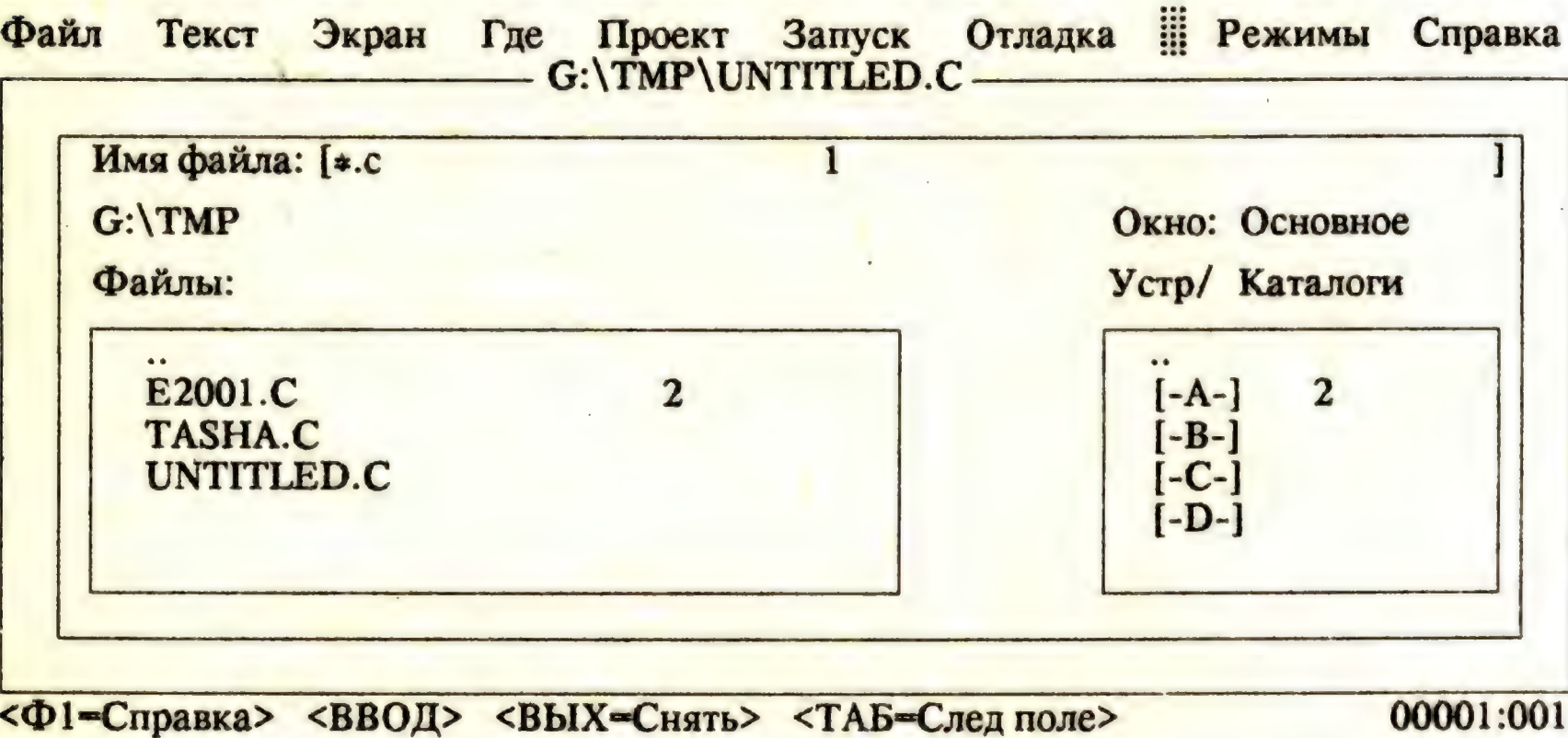
Таблица 14.1

Сокращенные наборы клавиш

Имя меню	Команда	Набор клавиш
Файл	Последний файл	Ф2
	Выход	АЛТ+Ф4
Текст	Отмена	АЛТ+ВШ
	В буфер	РЕГ+УД
	Копия	УПР+ВСТ
	Из буфера	РЕГ+ВСТ
	Очистка	УД
Экран	Экран вывода	Ф4
	Увеличить	УПР+Ф10
	Где	УПР+\<
Запуск	Заданный текст	Ф3
	Следующий повтор	РЕГ+Ф3
	Ошибка ниже	РЕГ+Ф4
	Ошибка выше	РЕГ+Ф5
	Рестарт	Ф5
	Пуск	Ф7
	До курсора	Ф8
	Шаг трассы	Ф10
	Шаг программы	Ф9
	Точка прерывания	Ф1
Отладка	Тема	РЕГ+Ф1
Справка	О справке	

14.3. КАДРЫ ДИАЛОГА

Среда Си выдает кадр диалога, когда ей нужна дополнительная информация или когда требуется подтвердить действие, которое выбрал пользователь. Кадр диалога содержит области, куда пользователь вводит информацию, выбирает режимы команд, управляет выполнением или аннулированием команды. Примеры кадров диалога команд "Открыть" и "Экран" приведены на рис. 31 и 32. Цифрами на этих рисунках отмечены различные области кадра диалога, которые поясняются ниже.



При появлении кадра диалога одна из областей кадра всегда имеет "фокус ввода". В этой области появляется мерцающее подчеркивание. Оно говорит о том, что вводимая с клавиатуры информация относится к данной области. Перемещать "фокус ввода" можно с помощью следующих клавиш:

- к следующей области — ТАБ;
- к предыдущей области — РЕГ+ТАБ;
- к любой области — АЛТ+ярко светящаяся буква в имени области.

Кадр диалога может содержать следующие области, отмеченные на рис. 31 и 32 соответствующими цифрами:

1 — Кадр текста — область экрана, в которую вводится информация для команд (например, имя файла). Если набираемый текст не вмещается в кадре текста, кадр текста прокручивается влево, чтобы освободить для ввода дополнительное место.

2 — Кадр списка — кадр, в котором перечислены объекты для выбора. Кадры списков представляют альтернативный способ выбора: вместо набора текста в кадре текста можно выбрать его в кадре списка. Выбрать нужный объект из списка можно либо перемещая фокус ввода к нужному объекту с помощью клавиш управления курсором, либо набрав первую букву в имени объекта. Выбираемый элемент нужно высветить нажатием длинной клавиши пробела. Чтобы выполнить команду с выбранным текстом, достаточно нажать ВВОД. На рис. 31 имеются два кадра списка: кадр списка файлов и кадр списка каталогов и устройств (отмечены цифрой 2).

3 — Командная кнопка — команда, заключенная в угловые скобки (< >), которая указывает возможное действие. Когда открывается кадр диалога, одна из командных кнопок отличается цветом угловых скобок. Такая кнопка называется *активной*. Действие, указанное в активной командной кнопке, будет выполняться по нажатию клавиши ВВОД. Чтобы сделать активной другую командную кнопку, необходимо переместить на нее фокус ввода. Выполнить действие, указанное в командной кнопке, имеющей фокус ввода, можно нажатием клавиши ВВОД или длинной клавиши пробела. После выполнения команды кадр диалога исчезает.

4 — Кадр "галочек" — область экрана, содержащая режимы, которым предшествуют левая и правая квадратные скобки ([]), позволяющие выбрать один или несколько режимов из группы. Перемещать фокус ввода вперед и назад между кадрами "галочек" можно с помощью комбинации клавиш ТАБ и РЕГ+ТАБ. Для включения кадра "галочки", имеющего фокус ввода, надо нажать длинную клавишу пробела. Во включенном кадре "галочки" появляется X.

5 — Круговые кнопки — область экрана, содержащая режимы, которым предшествуют левая и правая круглые скобки (). Можно выбрать только один режим из группы. Выбранная круговая кнопка помечается точкой внутри круглых скобок. При перемещении фокуса ввода к группе круговых скобок он всегда устанавливается на выбранной круговой кнопке из группы. Нужная кнопка выбирается с помощью клавиш управления перемещением курсора ↑ или ↓.

Глава 15

СОЗДАНИЕ И РЕДАКТИРОВАНИЕ
ИСХОДНЫХ ПРОГРАММ

Первым действием при разработке программ является создание и сохранение программы. Для создания новых программ и редактирования уже существующих в среде Си есть встроенный текстовый редактор, содержащий богатый набор операций редактирования. Эти операции наряду с командами меню "Файл", "Текст" и "Где" дают возможность создавать, сохранять и редактировать программы в среде Си.

15.1. ОПЕРАЦИИ ВСТРОЕННОГО ТЕКСТОВОГО РЕДАКТОРА

Встроенный текстовый редактор среды Си позволяет быстро выполнять такие операции редактирования, как перемещение курсора, вставка, выбор, копирование и удаление текста. Многие операции схожи с командами редактора WS.

Команды клавиатуры, выполняющие функции редактирования, приведены в табл 15.1.

Таблица 15.1

Команды редактирования в среде Си

Действие	Клавиши редактора среды	Альтернативные клавиши редактора WS
Управление курсором Сдвиг на символ влево Сдвиг на символ вправо Сдвиг на слово влево Сдвиг на слово вправо Сдвиг на строку вверх Сдвиг на строку вниз	← → УПР+← УПР+→ ↑ ↓	УПР+S УПР+D УПР+A УПР+F УПР+E УПР+X
К первому отступу текущей строки К началу текущей строки К первому отступу следующей строки К концу строки	НАЧ УПР+ВВОД КОН	УПР+Q S УПР+J УПР+Q D
К началу окна К концу окна		УПР+Q E УПР+Q X
К началу файла К концу файла Установить маркер 'п' Подвести к маркеру 'п'	УПР+НАЧ УПР+КОН	УПР+Q R УПР+Q C УПР+K п УПР+Q п
Прокрутить на: строку вверх строку вниз страницу вверх страницу вниз	УПР+↑ УПР+↓ СТР↑ СТР↓	УПР+W УПР+Z УПР+R УПР+C

Продолжение табл. 15.1

Действие	Клавиши редактора среды	Альтернативные клавиши редактора WS
одно окно влево одно окно вправо	УПР+СТР↑ УПР+СТР↓	
Выбор текста Символ слева Символ справа Слово слева Слово справа	РЕГ+← РЕГ+→ РЕГ+УПР+← РЕГ+УПР+→	
Текущую строку Строку выше Экран выше Экран ниже Начало файла Конец файла	РЕГ+↓ РЕГ+↑ РЕГ+СТР↑ РЕГ+СТР↓ РЕГ+УПР+НАЧ РЕГ+УПР+КОН	
Вставка и копирование Режим вставки включить или выключить	ВСТ	УПР+V
Копировать текст в буфер без удаления Копировать текст в буфер с удалением Копировать текущую строку в буфер с удалением Копировать в буфер с удалением конца строки	УПР+ВСТ РЕГ+УД	УПР+Y УПР+Q Y
Вставить текст из буфера Вставить пустую строку ниже Вставить пустую строку выше Разорвать строку в позиции курсора Вставить управляющий символ	РЕГ+ВСТ КОН ВВОД УПР+М УПР+Р, затем УПР+символ	НАЧ+УПР+N
Удаление Удалить символ слева от курсора Удалить текущий символ Удалить правую часть текущего слова	ВШ УД	УПР+N УПР+G УПР+T
Удалить выбранный текст Удалить ведущие пробелы в выбранном тексте	УД РЕГ+ТАБ	УПР+G
Отменить все изменения	АЛТ+ВШ	
Поиск Искать текст Искать выбранный текст Повторить поиск Искать текст и изменить Найти парную скобку		УПР+Q F УПР+\ РЕГ+ФЗ УПР+Q A УПР+]

Текстовый редактор интегрированной среды Си работает в одном из двух режимов: режиме вставки и режиме замены. Когда редактор находится в режиме вставки, все набираемые символы вставляются

слева от курсора. Курсор в этом режиме представляет собой мерцающее подчеркивание.

Когда редактор находится в режиме замены, то набираемые символы замещают символы в текущей позиции. В режиме замены курсор представляется блоком. Для переключения редактора из режима вставки в режим замены и наоборот используются клавиши ВСТ или УПР+V.

Используя редактор интегрированной среды Си, можно легко найти парные левую и правую фигурные скобки {}, квадратные скобки [], угловые скобки, изображаемые знаками меньше и больше <>, или круглые скобки (). Чтобы найти парную фигурную, квадратную, угловую или круглую скобку, надо выполнить следующее:

1) переместить курсор на фигурную, квадратную, угловую или круглую скобку, для которой требуется найти парную скобку;

2) нажать клавиши УПР+].

При редактировании большого файла в различных местах этого файла можно установить маркеры места, а затем переходить на нужный маркер для редактирования соответствующей части файла. Для любого файла можно использовать до четырех маркеров места, пронумерованных от 0 до 3.

Встроенный текстовый редактор допускает использование манипулятора графической информации для передвижения курсора и выбора текста.

Чтобы передвинуть курсор манипулятором, надо разместить указатель манипулятора там, где должен быть курсор, и щелкнуть левой кнопкой.

Ниже описываются способы выбора текста манипулятором.

Если при выборе текста была допущена ошибка, надо щелкнуть у выбранного текста для отмены выбора.

Выбор текста манипулятором

Объект выбора	Выполнение
Символ	Подвести указатель к символу, расположенному справа от символа, который надо выбрать, нажать на левую кнопку и протянуть указатель влево над символом или символами, которые нужно выбрать. Протягивание вправо выбирает символ, на котором установлен курсор
Слово	Дважды щелкнуть у этого слова
Строка	Подвести указатель к первой позиции строки, нажать левую кнопку, протянуть яркость вниз на одну строку и отпустить кнопку. Протягивание вверх выбирает строку над курсором
Несколько строк	Подвести указатель к первой из выбираемых строк, нажать левую кнопку и протянуть указатель вверх или вниз через количество строк, которые нужно выбрать. В результате выделенные строки будут ярко высвечены

15.2. МЕНЮ "ТЕКСТ"

Команды меню "Текст" показаны на рис. 33 и предназначены для работы с выбранным текстом. Они дают возможность перемещать, копировать, удалять текст, а также аннулировать изменения текущей строки.

Текст	
Отмена	АЛТ+ВШ
В буфер	РЕГ+УД
Копия	УПР+ВСТ
Из буфера	РЕГ+ВСТ
Очистка	УД
Только чтение	

Рис. 33. Меню "Текст"

Команда "Отмена" отменяет все изменения текущей строки. Строка восстанавливается при условии, что курсор находится на строке. Если курсор сдвинуть со строки, то изменения становятся постоянными, и их нельзя аннулировать. Сокращенный набор клавиш, соответствующий этой команде, — АЛТ+ВШ.

Этой командой нельзя восстанавливать строки, удаленные комбинацией клавиш УПР+У. В этом случае для восстановления надо использовать комбинацию РЕГ+ВСТ.

Команда "В буфер" удаляет выбранный текст и помещает его в буфер, представляющий область временной памяти, которую встроенный текстовый редактор среды Си использует, чтобы сохранить копию удаленного текста. Предыдущее содержимое буфера удаляется. Этой команде соответствует сокращенный набор клавиш РЕГ+УД.

Команда "Копия" копирует выбранный текст в буфер, не удаляя его из исходного файла. Предыдущее содержимое буфера удаляется. Сокращенный набор клавиш для этой команды — УПР+ВСТ.

Команда "Из буфера" вставляет текст из буфера слева от курсора, если в момент выполнения команды нет выбранного текста. Если есть выбранный текст, он заменяется на текст из буфера.

Текст в буфере остается до конца сеанса в среде Си, если не выполняются команды, сохраняющие информацию в буфере (например, "Копия" или "В буфер"). Поэтому текст из буфера можно вставлять столько раз, сколько потребуется.

Команда "Очистка" удаляет выбранный текст из исходного файла, не помещая в буфер. Сокращенный набор клавиш — УД.

Команда "Только чтение" включает и выключает режим только чтения. Если этот режим установлен, то в строке подсказки появляется буква "Ч" и в текущий загруженный файл нельзя внести никакие изменения. Эта команда полезна во время отладки, чтобы исключить случайные изменения файла.

15.3. МЕНЮ "ГДЕ"

Меню "Где" показано на рис. 34.

Где	
Искать...	
Заданный текст	УПР+\
Следующий повтор	Ф3
Изменить...	
Функция	
Ошибка ниже	РЕГ+Ф3
Ошибка выше	РЕГ+Ф4

Рис. 34. Меню "Где"

Команды "Искать", "Заданный текст" и "Следующий повтор" используются для поиска текста в исходном файле.

Командой "Искать" можно найти первое появление символа, слова, группы символов или слов. Текст для поиска задается в кадре диалога, который открывается при выборе команды "Искать". Команду "Искать" можно выполнить, используя сокращенный набор клавиш УПР+Q F.

Другим способом поиска текста в программе является команда "Заданный текст". Если перед использованием команды "Заданный текст" в программе был выбран текст с помощью клавиш выбора текста, то в результате выполнения команды "Заданный текст" будет высвечено следующее появление выбранного текста. Команде "Заданный текст" соответствует сокращенный набор клавиш УПР+\. Если выбранного текста в программе нет, то действия команды "Заданный текст" будут такие же, как и действия команды "Искать".

Примечание. При выборе текста для поиска надо помнить, что выбранный текст должен располагаться на одной строке.

Команда "Следующий повтор" ищет текст, который был определен в предыдущей команде "Искать", "Заданный текст" или "Изменить". Этой команде соответствует функциональная клавиша Ф3 или сокращенный набор клавиш УПР+L.

Команда "Изменить" используется для замены одного текста другим. Если выбрана команда "Изменить", то появляется кадр диалога, запрашивающий дополнительную информацию (рис. 35).

Найти:	[			]
Изменить на	[			]
[]	Слово			
[]	Прописные/строчные			
[]	Регулярное выражение			
<Найти и проверить>		< Изменить >	< Снять >	<Справка>

Рис. 35. Кадр диалога команды "Изменить"

В этом кадре диалога надо указать заменяемый и заменяющий тексты. Кроме этого, можно выбрать командную кнопку "Изменить" для выполнения замены всех повторений указанного текста, не запрашивая подтверждения каждого изменения. Если выбирается командная кнопка "Найти и проверить", после того как нужный текст найден, выдается запрос на дальнейшие действия в виде следующего кадра диалога (рис. 36).

Заменить это вхождение?			
< Да >	< Нет >	< Снять >	<Справка>

Рис. 36. Кадр подтверждения

Команда "Функция" из меню "Где" позволяет найти точку входа заданной функции. Перед использованием этой команды файл должен быть прокомпилирован с режимом "Отладка". Имя функции вводится в кадр диалога, который появляется после выбора команды "Функция".

Команды "Ошибка ниже" и "Ошибка выше" используются для просмотра списка синтаксических ошибок, обнаруженных во время компиляции. Если во время компиляции были найдены ошибки, то открывается окно "Ошибки" в нижней части экрана, в котором выводится список обнаруженных ошибок компиляции. При этом курсор в основном окне указывает на строку исходного текста, в которой обнаружена первая ошибка из списка. По команде "Ошибка ниже" курсор перемещается на строку программы, где обнаружена следующая ошибка из списка, а по команде "Ошибка выше" курсор перемещается на строку программы, где обнаружена предыдущая ошибка из списка.

Просмотреть список ошибок можно также следующим способом:

- с помощью функциональной клавиши Ф6 переместить курсор в окно "Ошибки";
- клавишами управления курсором выбрать интересующее сообщение об ошибке.

Строка исходной программы, которая вызвала данную ошибку, ярко высветится. Для получения справочной информации об этом сообщении достаточно нажать клавишу Ф1 или щелкнуть правой кнопкой манипулятора.

15.4. МЕНЮ "ФАЙЛ"

Команды меню "Файл" управляют исходными файлами. Они используются для создания новых или загрузки существующих файлов, сохранения вновь созданных и отредактированных файлов, слияния файлов и печати файла или выбранного текста. Кроме этого, команды меню "Файл" организуют выход из среды Си. Меню "Файл" приведено на рис. 37.

Файл	
Новый	
Открыть...	
Последний файл	Ф2
Слить...	
Сохранить	
Сохранить как...	
Сохранить все	
Печать...	
ДОС	
Выход	АЛТ+Ф4

Рис. 37. Меню "Файл"

Команда "Новый" используется для создания нового файла в основной памяти. По этой команде очищается память и основное окно. Если текущим окном во время выполнения команды "Новый" было окно "Блокнот", то очищается это окно. Имя файлу назначается при сохранении файла.

Команда "Открыть" загружает существующий файл для редактирования. После выбора команды "Открыть" открывается кадр диалога, показанный на рис. 31.

Чтобы загрузить файл из текущего каталога, достаточно ввести имя файла в кадре текста "Имя файла" или выбрать имя файла в кадре списка "Файлы".

Чтобы загрузить файл из другого каталога, можно либо ввести имя каталога и имя файла в кадре текста "Имя файла", либо сначала выбрать каталог в кадре списка "Устр/Каталоги", а затем имя файла в кадре списка "Файлы". Чтобы выбрать имя файла из списка файлов с помощью манипулятора графической информации, достаточно дважды щелкнуть левой кнопкой манипулятора у имени файла в кадре списка.

Команда "Последний файл" полезна, если в одном сеансе работы в среде Си редактируются два исходных файла. После того, как один исходный файл сохранен и загружен другой исходный файл для редактирования, можно использовать команду "Последний файл" для загрузки ранее сохраненного файла. Если к моменту выполнения этой команды в текущий файл были внесены изменения, будет выдан запрос о том, надо ли сохранять текущий файл. Команде "Последний файл" соответствует функциональная клавиша F2.

Команда "Слить" вставляет указанный файл после строки, на которую указывает курсор в окне редактирования. Имя файла для слияния указывается в кадре диалога, который открывается по команде "Слить". Этот кадр такой же, что и для команды "Открыть".

Команды "Сохранить" и "Сохранить как" сохраняют отредактированный файл на диске. Команда "Сохранить" используется для сохранения файла с его исходным именем.

Если сохраняется новый, еще не названный файл, то появляется кадр диалога, запрашивающий имя файла. Такой же кадр диалога появляется после выбора команды "Сохранить как" для сохранения файла с изменением имени.

Команда "Сохранить все" позволяет сохранить все измененные файлы, включая файл из основного окна, файл из окна "Блокнот", программный список, файл истории отладки.

Командой "Печать" можно распечатать или весь редактируемый файл, или выбранную часть этого файла. Файлы печатаются без деления на страницы и без заголовков.

Чтобы воспользоваться этой командой, необходимо, чтобы печатающее устройство было присоединено к ПЕЧ1 (LPT1). Используя команду РЕЖИМ (MODE) операционной системы Альфа-ДОС, можно переназначить ПЕЧ1 (LPT1) на ПОСЛ1 (COM1).

По команде "ДОС" можно временно вернуться в среду команд Альфа-ДОС, где выполнить другие программы или команды Альфа-ДОС. Интегрированная среда Си остается в памяти. Надо отметить, что перед возвратом в среду Си нельзя удалять файлы с расширением .МАК, исходные файлы Си или файлы из текущего программного списка.

Для выполнения команды "ДОС" среде Си должен быть доступен файл COMMAND.COM. Файл COMMAND.COM среда Си вначале

ищет в каталоге, заданном переменной среды COMSPEC, а затем в текущем каталоге. Для возврата в среду Си нужно ввести команду **ВЫХОД (EXIT)** Альфа-ДОС.

Команда **"Выход"** удаляет интегрированную среду Си из основной памяти и обеспечивает возврат в Альфа-ДОС (сокращенный набор клавиш **ALT+Ф4**). Если файл редактировался, но не был сохранен, то будет выдан запрос о том, надо ли сохранять этот файл. Чтобы сохранить программу, надо нажать клавишу **ВВОД** или выбрать ответ **"Да"**. Если программа не имеет имени, то появится кадр диалога **"Сохранить как"**. Чтобы вернуться в операционную систему без сохранения файла, надо нажать **"Н"** или выбрать командную кнопку **"Нет"**.

Глава 16

КОМПИЛЯЦИЯ, ВЫПОЛНЕНИЕ И ОТЛАДКА ПРОГРАММ

После создания и сохранения программы следующими этапами разработки являются компиляция, выполнение и отладка этой программы. Для этих целей в среде Си имеются встроенный компилятор и отладчик. Для управления компиляцией, выполнением и отладкой в среде Си используются команды меню **"Режимы"**, **"Проект"**, **"Запуск"**, **"Экран"** и **"Отладка"**.

16.1. МЕНЮ "РЕЖИМЫ"

Открытое меню **"Режимы"** показано на рис. 38.

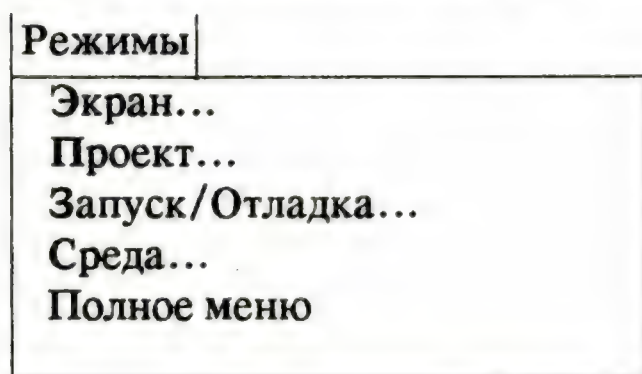


Рис. 38. Меню **"Режимы"**

Команда **"Экран"** используется для изменения режимов среды Си. Выбирая круговые кнопки и кадры галочек в кадре диалога команды **"Экран"** (см. рис. 32), можно изменить цвета текста, фона окон и границ; убрать полосы прокрутки, если не используется манипулятор графической информации; установить нужный размер табуляции (по умолчанию 8).

С помощью команды "Проект" устанавливаются режимы компиляции и редактирования программ в среде Си. Кадр диалога команды "Проект" показан на рис. 39.

Выбрать режимы

☐ Отладка

☐ Прогон

Настроить режимы

< Компилятор >

< Редактор LINK >

< OK >

< Снять >

< Справка >

Рис. 39. Кадр диалога команды "Проект"

Выбирая круговую кнопку, можно построить исполняемый файл для последующей отладки либо для выполнения (прогона). Для этого служит группа кнопок "Выбрать режимы".

Командной кнопкой "Компилятор" можно установить различные режимы компиляции, такие как модель памяти, уровень выдачи предупреждающих сообщений, запрещение расширений, построение отладочной информации, проверка указателей и стека, оптимизация (рис.40).

ОБЩИЕ

Модель памяти

☐ Малая

☐ Средняя

☐ Компактная

☐ Большая

☐ Сверхбольшая

Ошибки

☐ Ур 0

☐ Ур 1

☐ Ур 2

☐ Ур 3

Язык Си

☐ Совместимость с АНИС

☐ Расширения

ДЛЯ ОТЛАДКИ

☐ Для CV

☐ Только номера строк

☒ Проверка указателя

☒ Компиляция изменений

СПЕЦИАЛЬНЫЕ

Оптимизация

☐ Вкл

☐ Выкл

☐ Полная

☒ Проверка стека

DEFINE: []

<OK>

<Снять>

<Справка>

Рис. 40. Кадр режимов компилятора

Командная кнопка "Редактор LINK" позволяет установить режимы редактирования, такие как различать или не различать прописные и строчные буквы, установить размер стека, требовать новый диск

перед записью исполняемого файла, строить отладочную информацию для многооконного отладчика и отладчика среды Си.

Команда "Запуск/Отладка" используется, если для выполнения программы необходимо задать аргументы, а также для управления скоростью трассировки, сменой экрана и созданием истории отладки.

Команда "Среда" выполняется, если необходимо определить каталоги для поиска файлов, включаемых по директиве препроцессора `#include`, библиотек и файлов со справочной информацией. По умолчанию маршруты поиска определяются переменной среды или текущим каталогом, если переменная среды не определена.

Среда Си имеет два набора команд меню — полные и короткие меню. Короткие меню обычно используются начинающими пользователями среды Си для разработки одномодульных программ. Они содержат самые необходимые команды. Например, короткое меню "Режимы" показано на рис. 41, в отличие от полного, представленного на рис. 38.

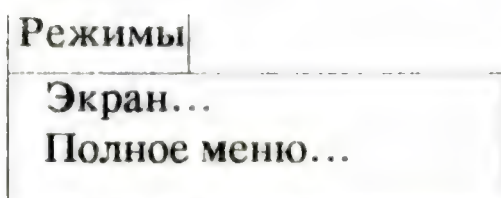


Рис. 41. Короткое меню "Режимы"

Полные меню обеспечивают доступ ко всем имеющимся в среде Си возможностям, включая программные списки и средства отладки. Для переключения полных и коротких меню используется команда "Полное меню".

16.2. МЕНЮ "ПРОЕКТ"

Для того чтобы откомпилировать и отредактировать программу, надо использовать команды меню "Проект" (рис. 42).

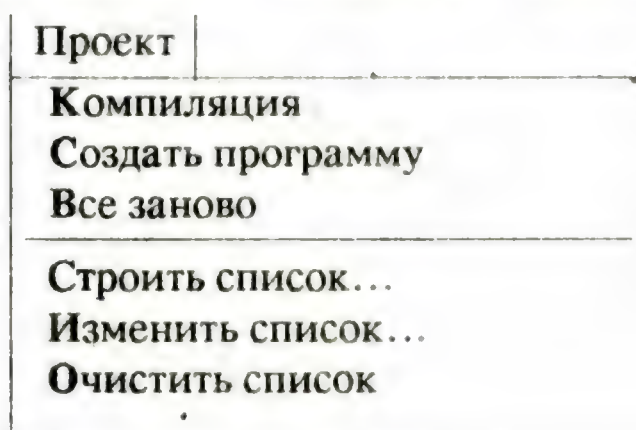


Рис. 42. Меню "Проект"

Программа, разрабатываемая в среде Си, может состоять из одного или нескольких исходных файлов. Соответственно она называется *одномодульной* либо *многомодульной* программой.

При разработке проекта, представляющего многомодульную программу, всегда используется программный список — файл, содержащий список модулей, составляющих Си-программу. Кроме исходных файлов, в программный список могут быть включены объектные файлы и автономные библиотеки (с расширением .LIB).

При компиляции многомодульной программы программный список просматривается программными средствами интегрированной среды Си, чтобы определить, изменялись ли какие-либо модули после последней компиляции программы. Если такие модули есть, перед повторным построением программы автоматически выполняется их компиляция. Этот механизм гарантирует, что построенная программа всегда актуальна, т.е. включает все последние изменения модулей.

Компилятор сохраняет программный список в файле на диске с таким же основным именем, как и основное имя программы, и расширением .MAK.

При разработке одномодульных программ строить программный список не требуется.

Команда "Компиляция" создает объектный файл из текущего исходного файла. Обычно эта команда используется для компиляции одномодульной программы.

Для компиляции и редактирования всех измененных модулей многомодульной программы надо использовать команду "Создать программу". Измененным считается модуль, который был скорректирован после последней компиляции либо компилировался после последнего редактирования.

Команда "Все заново" компилирует и редактирует все модули многомодульной программы, даже если они не корректировались.

Для обслуживания программных списков используются команды "Строить список", "Изменить список" и "Очистить список".

Команда "Строить список" создает новый программный список или загружает существующий программный список в интегрированную среду Си.

После выбора команды "Строить список" открывается кадр диалога, такой же как и для команды "Открыть", но в кадре списка будут перечислены все файлы с расширением .MAK.

Если ввести имя несуществующего файла с расширением .MAK, автоматически открывается кадр диалога команды "Изменить список" (рис. 43) и можно добавить модули в новый программный список и сохранить его.

После загрузки программного списка вся работа в среде Си идет под управлением этого программного списка.

Команда "Очистить список" удаляет из памяти текущий программный список. После выполнения этой команды компилятор среды Си предполагает, что все компилируемые исходные файлы являются одномодульными программами.

Команда "Изменить список" используется для обновления загруженного программного списка. После выбора команды "Изменить список" открывается кадр диалога, показанный на рис. 43.

Для добавления файла к программному списку надо выполнить следующую последовательность действий:

- либо ввести имя файла в кадре текста и нажать ВВОД;
- либо высветить имя в кадре списка "Файлы" и выбрать командную кнопку "Доб/Удалить". Для удаления файла из программного списка надо высветить файл в кадре списка "Прогр список" и выбрать командную кнопку "Доб/Удалить".

Имя файла: [*.*]
G:\TMP
Файлы:

E2001.C	TASHA.C
E2115.C	UNTITLED.C
SAM.C	
SUMMA.C	

Устр/Каталоги:

..
[-A-]
[-B-]
[-C-]

Прогр список: TASHA.MAK

--

<Доб/Удалить>
<Очистить>

<Сохранить> <Снять> <Справка>

Рис. 43. Кадр диалога команды "Изменить список"

Чтобы удалить все файлы из программного списка, достаточно выбрать командную кнопку "Очистить".

Команда "Очистить список" не изменяет содержимое программного списка (файла с расширением .МАК), а только удаляет его из памяти.

16.3. МЕНЮ "ЗАПУСК"

Команды меню "Запуск" (рис. 44) управляют загрузкой и выполнением программы.

Запуск	
Рестарт	РЕГ+Ф5
Пуск	Ф5
До курсора	Ф7
Шаг трассы	Ф8
Шаг программы	Ф10
Трасса	

Рис. 44. Меню "Запуск"

Команда "Пуск" начинает или продолжает выполнение программы. Программа будет выполняться от текущего оператора до тех пор, пока не будет:

- достигнута точка прерывания;
- выполнено условие, заданное в условном прерывании;
- достигнут конец программы;
- выполнение программы прервано пользователем комбинацией клавиш УПР+С или УПР+СТОП.

Команда "Рестарт" перезагружает программу в память и сбрасывает текущие значения всех переменных. Эта команда полезна при необходимости повторного запуска в одной из следующих ситуаций:

- во время отладки в пошаговом режиме;
- во время приостановки программы в точке прерывания;
- после принудительного прерывания выполнения программы пользователем по нажатию клавиш УПР+СТОП.

Команда "Рестарт" фактически не начинает выполнение программы. Для этого надо использовать одну из команд "Пуск", "До курсора", "Шаг трассы" или "Шаг программы".

Команда "До курсора" выполняет программу, начиная с текущего оператора, до оператора, на который указывает курсор. Прерывание происходит перед выполнением этого оператора. Этой команде соответствует сокращенный набор клавиш — Ф7.

Команды "Шаг трассы" и "Шаг программы" используются для пошагового выполнения программы. Эти команды могут выполняться с помощью сокращенных наборов клавиш, соответственно Ф8 и Ф10.

Разница между этими командами заключается в следующем:

команда "Шаг трассы" трассирует все вызываемые функции, т.е. если текущим оператором является вызов функции, то выполнение команды "Шаг трассы" вызывает переход к первому оператору в функции. Таким образом можно отлаживать функцию;

команда "Шаг программы" выполняет трассировку программы, обходя все вызываемые функции, т.е. если текущим оператором является вызов функции, то выполнение команды "Шаг программы" вызывает переход к оператору, следующему за вызовом функции. Команда "Шаг программы" полезна, если вызываемая функция уже отлажена и работает правильно.

Команда "Трасса" включает высвечивание каждого выполняемого оператора. Сама команда не начинает выполнение программы. После выбора команды "Трасса" для запуска программы необходимо использовать одну из команд "Пуск", "До курсора", "Шаг программы". Если установлено высвечивание (была выполнена команда "Трасса"), для прерывания выполнения программы достаточно нажать любую клавишу. Скорость трассировки может быть установлена выбором соответствующего режима в команде "Запуск/Отладка" меню "Режимы".

16.4. МЕНЮ "ОТЛАДКА"

Команды меню "Отладка" (рис. 45) в сочетании с командами меню "Запуск" управляют отладкой в среде Си.

Команды меню "Отладка" по выполняемым функциям разделяются на три группы (на рис. 45 они разделены горизонтальными линиями).

К первой группе относятся команды, управляющие прерываниями.

Команда "Точка прерывания" (сокращенный набор клавиш Ф9) используется для установки и удаления безусловных точек прерывания. Точка прерывания вызывает паузу в выполнении программы, и управление передается пользователю.

Отладка	
Вызов...	
Точка прерывания	Ф9
Когда не ноль...	
Следить за...	
Изменить...	
История	
Отменить	
Воссоздать	
Удалить ввод	

Рис. 45. Меню "Отладка"

В среде Си существуют два способа установки безусловных точек прерывания:

- поместить курсор на строку, перед которой надо установить точку прерывания, и нажать Ф9;
- выбрать команду "Точка прерывания" в меню "Отладка". В открывшийся кадр диалога ввести имя файла и номер нужной строки в виде:

имя-файла:номер-строки

Все установленные точки прерывания будут перечислены в кадре списка кадра диалога команды "Точка прерывания".

Все строки, на которых установлены точки прерывания, будут ярко высвечены. Если после установки точек прерывания нажать Ф5 или выбрать команду "Пуск" из меню "Запуск", то программа будет выполняться до следующей точки прерывания.

Для удаления отдельной точки прерывания достаточно поместить курсор на нужную строку и нажать клавишу Ф9. Для удаления всех установленных безусловных точек прерывания надо выбрать командную кнопку "Очистить" кадра диалога команды "Точка прерывания".

Команда "Когда не нуль" устанавливает условные точки прерывания при отладке программы. Эти точки носят название точек наблюдения. После выбора команды появляется кадр диалога, запрашивающий выражение. После выполнения каждого оператора будет вычисляться введенное выражение. Выполнение программы прерывается, когда выражение становится истинным (не равным нулю). Например, если в кадре диалога команды "Когда не нуль" было введено выражение $i > 90$, программа будет выполняться без остановки до тех пор, пока значение переменной i не превысит 90.

Точки наблюдения остаются в памяти до тех пор, пока не будут удалены. Чтобы удалить точку наблюдения, надо выбрать нужное выражение из кадра списка "Список" и нажать ВВОД. Для удаления всех точек наблюдения достаточно выбрать командную кнопку "Очистить". Кадр диалога команды "Когда не нуль" приведен на рис. 46.

Выражение: [	]
Список:	
<Доб/Удалить> <Очистить> <Снять> <Справка>	

Рис. 46. Кадр диалога команды "Когда не нуль"

Команда "Вызов" используется для просмотра цепочки вызовов функций во время отладки программы. При этом первой функцией в списке будет функция, вызванная самой последней. Каждая функция выдается с переданными ей аргументами. Если выбрать одну из функций, перечисленных в списке, то в основном окне отобразится текст этой функции, а курсор будет указывать на оператор, который должен выполняться после возврата управления этой функции.

Следующая группа команд предназначена для контроля за выражениями и изменением значений переменных.

Командой "Следить за" определяются выражения, за которыми необходимо установить контроль, так называемые выражения наблюдения. Кадр диалога команды "Следить за" такой же, как и на рис. 46. Выражение для наблюдения вводится в кадре текста "Выражение" и может быть любым допустимым в Си выражением.

Примеры.

счет

счет - 8

$i == 4$

Все выражения наблюдения помещаются в окно "Отладка", которое автоматически открывается в верхней части экрана, если была выбрана команда "Следить за". При выполнении программы по мере изменения значений выражений обновляется вывод в окне "Отладка". При этом выполнение программы не прерывается.

Обычно значения переменных в окне "Отладка" появляются в формате, принятом по умолчанию для их типа. Для выдачи значения переменной в другом формате достаточно вслед за именем переменной через запятую указать одну из букв, описанных ниже.

Спецификация формата вывода значений

Символ	Формат вывода
d	Десятичное целое со знаком
i	Десятичное целое со знаком
u	Десятичное целое без знака
o	Восьмеричное целое без знака
x	Шестнадцатеричное целое
f	Величина со знаком в десятичном формате с плавающей точкой с шестью десятичными позициями
e	Величина со знаком в экспоненциальном формате с шестью десятичными позициями (конечные нули и десятичная точка усекается)
g	Величина со знаком в десятичном формате с плавающей точкой (f) или экспоненциальном формате (e), в зависимости от того, который из них более компактен
c	Один символ
z	Структура с именами полей

Допустим, например, что в отлаживаемой программе имеется переменная, объявленная следующим образом:

```
struct {
    int a;
    char b;
} str;
```

Для наблюдения были введены две переменные :

```
str
str,z
```

Тогда в окне отладки появятся следующие строки:

```
str : { 4, '%' }
str.a : 4
str.b : '%'
```

Для вывода массива символов в виде строки надо задать выражение в следующем виде:

```
&имя-массива,s
```

Команда "Изменить" используется для изменения значений простых переменных. Имя переменной и новое значение вводятся в соот-

ветствующие кадры текста кадра диалога команды "Изменить". Чтобы изменить значение структуры или массива, надо изменять отдельно значение каждого элемента.

Третья группа команд меню "Отладка" предназначена для сохранения и последующего использования протокола отладки (истории отладки). Эта возможность может быть полезна для сохранения действий пользователя во время одного сеанса отладки с целью воспроизведения в следующем сеансе. История отладки содержит два файла:

- файл, содержащий все команды, задаваемые отладчику (файл с расширением .HIS);
- файл, содержащий весь ввод для программы, введенный с клавиатуры (файл с расширением .INP).

Включить ведение истории отладки можно, выбрав команду "История". Если история отладки включена, то в меню рядом с ней появляется точка. История отладки всегда ведется с начала выполнения программы. Если команда "История" была выбрана в процессе отладки, то после предупреждающего сообщения выполнение программы будет прервано и произведен рестарт программы.

Если при отладке программы была включена история отладки, то пользователь имеет возможность отменить последнюю команду отладчика. Для этого достаточно выбрать команду "Отменить". С помощью этой команды можно вернуться к началу сеанса отладки, отменяя по одной команде команды отладки.

Другой возможностью, доступной пользователю при ведении истории отладки, является возможность воспроизведения всего или части проведенного сеанса. Для этого используется команда "Воссоздать". После выбора команды "Воссоздать" производится рестарт программы, а затем выполняются все отладочные команды из файла .HIS с использованием файла .INP, когда требуется ввод с клавиатуры.

Для воспроизведения части сеанса отладки сначала надо воспользоваться набором клавиш РЕГ+Ф8 или РЕГ+Ф10. Набор клавиш РЕГ+Ф8 используется для выдачи предыдущей точки прерывания программы, а РЕГ+Ф10 для выдачи следующей точки прерывания в истории отладки. С помощью этих клавиш можно продвинуться на любую точку в истории отладки. Если затем выбрать команду "Воссоздать", то среда Си снова выполнит сеанс с начала до установленной точки.

Часто бывает удобно повторить сеанс отладки, используя другую информацию, вводимую с клавиатуры. Для этого надо воспользоваться командой "Удалить ввод", которая позволяет удалить весь ввод или часть пользовательского ввода и заменить его новым.

Для замены всего ввода пользователя надо выполнить следующие действия:

- сделать рестарт программы (команда "Рестарт" меню "Запуск" или сокращенный набор клавиш РЕГ+Ф5);
- выбрать команду "Удалить ввод".

При последующем выполнении команды "Воссоздать" программа будет ожидать ввода пользователя.

Для удаления части пользовательского ввода необходимо:

- для продвижения к точке, в которой начнется новый ввод, использовать клавиши РЕГ+Ф8 или РЕГ+Ф10;
- выбрать команду "Воссоздать" для повторения сеанса отладки до этой точки;
- выбрать команду "Удалить ввод". Среда Си сохранит ввод из файла .INP, который уже был использован, и удалит весь оставшийся.

16.5. МЕНЮ "ЭКРАН"

Команды меню "Экран" помогают пользователю получить дополнительную информацию во время работы в среде Си при редактировании, компиляции, отладке. Меню "Экран" показано на рис. 47.

Экран
Список Include
Экран вывода Ф4 Увеличить УПР+Ф10 Окна

Рис. 47. Меню "Экран"

Команда "Список" полезна при разработке многомодульной программы. С ее помощью можно легко выбрать нужный модуль для просмотра. Эта команда выполняется при наличии программного списка, загруженного командой "Строить список" меню "Проект", и обеспечивает быстрый способ перемещения между модулями без использования команды "Открыть". Использовать команду "Список" удобнее, чем команду "Открыть", по следующим причинам:

- команда "Список" выдает только файлы, которые входят в программный список; команда "Открыть" может выдать значительно больше файлов;
- команда "Список" выдает файлы программного списка, которые зарегистрированы в других каталогах; команда "Открыть" выдает только файлы текущего каталога.

После выбора команды "Список" открывается кадр диалога, в котором перечислены все модули текущей программы. С его помощью можно выбрать нужный модуль для загрузки его в текущее окно просмотра. Это может быть либо основное окно, либо окно "Блокнот".

Аналогично команде "Список" команда "Include" используется для загрузки нужного файла включения для его просмотра и редактирования.

Другие команды меню "Экран" управляют окнами среды Си.

Команда "Экран вывода" (сокращенный набор клавиш Ф4) используется для переключения между экраном среды Си и экраном вывода программы. Эта команда позволяет просмотреть вывод программы во время отладки.

С помощью команды "Окна" можно открыть, закрыть или переключить окна среды Си. В среде Си пользователь может использовать следующие окна: Основное, Блокнот, Отладка, Справка, Локальные, Регистры, Вывод, Ошибки. Эти окна описаны ниже.

Некоторые окна среды Си используют одну и ту же область экрана и поэтому называются *разделяемыми*. К разделяемым окнам относятся: "Справка" и "Отладка"; "Локальные" и "Регистры"; "Блокнот" и "Вывод".

Основное окно всегда открыто, его нельзя закрыть никакими командами.

Окна среды Си

Окно	Использование
Основное	Основное окно среды Си для редактирования файлов. В этом окне редактируется, компилируется, выполняется и отлаживается исходная программа. В этом окне можно использовать все команды текстового редактора среды Си. Основное окно не может быть закрыто
Блокнот	Дополнительное окно среды Си для редактирования. В нем также можно использовать все команды текстового редактора, но программу из этого окна нельзя компилировать. Это окно используется для сохранения временной информации. Один и тот же файл не может быть загружен одновременно и в основное окно, и в окно "Блокнот"
Отладка	В этом окне выводятся выражения наблюдения. Выражения наблюдения в это окно добавляются либо с помощью команды "Следить за", либо набором выражения непосредственно в окне. В окне также можно пользоваться всеми командами текстового редактора
Справка	В этом окне выводится справка о языке Си, командах среды и библиотеке времени выполнения. В этом окне работают обычные команды текстового редактора только для чтения (например, Стр↑, Стр↓, копирование, поиск и т.п.)
Локальные	В это окно выводятся выражения наблюдения для всех локальных переменных активной в данный момент функции. Содержимое этого окна обновляется всякий раз, когда управление передается из функции к функции
Регистры	В это окно выводится содержимое регистров компьютера
Вывод	В это окно выводится копия экрана вывода программы. Это окно имеет режим только чтения. Из этого окна можно копировать в буфер ¹
Ошибки	Открывается автоматически для вывода сообщений об ошибках компилятора или редактора. В этом окне могут выполняться команды текстового редактора только для чтения. Чтобы найти строку, в которой произошла ошибка, надо выбрать сообщение об ошибке. Ошибочная строка ярко высветится. Для перехода к следующей ошибке надо использовать РЕГ+Ф3, к предыдущей ошибке — РЕГ+Ф4

¹Это окно не может использоваться для графического вывода

Некоторые окна открываются автоматически после выполнения определенных команд. Так, если была выполнена команда "Следить за" меню "Отладка", то автоматически открывается окно "Отладка", в котором отображается выражение наблюдения.

Если во время компиляции или редактирования программы были обнаружены ошибки, то открывается окно "Ошибки".

Окно "Справка" также открывается автоматически при выполнении команды "Тема" меню "Справка".

Для управления окнами, кроме команды "Окна", используются наборы клавиш.

Клавиши управления окнами	
Действие	Набор клавиш
Перемещение курсора между окнами	Ф6
Закрытие текущего окна	УПР+Ф4
Изменение размера текущего окна	УПР+Ф8
Переключение разделяемых окон	УПР+Ф6
Увеличение размера всех окон до максимального (команда "Увеличить")	УПР+Ф10

Команда "Увеличить" переключает размер окон с обычного на максимальный (сокращенный набор клавиш УПР+Ф10). При этом увеличивается размер не только активного окна, но и всех окон среды Си. Для перемещения из одного окна к другому можно использовать клавишу Ф6.

16.6. МЕНЮ "УТИЛИТЫ"

Команды меню "Утилиты" используются для выполнения команд Альфа-ДОС и часто используемых средств пользователя без выхода из среды Си. Меню "Утилиты" показано на рис. 48.

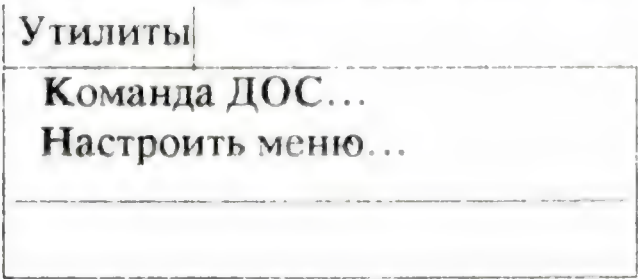


Рис. 48. Меню "Утилиты"

"Команда ДОС" используется для выполнения отдельной команды Альфа-ДОС. При выполнении этой команды открывается кадр диалога. Имя программы и аргументы должны быть введены в кадре текста. Эта программа должна быть в одном из каталогов, перечисленных в переменной МАРШ (PATH) среды.

Пользователь имеет возможность добавить собственные команды (до 8 команд) в меню "Утилиты". Для этой цели используется

команда "Настроить меню". С помощью этой команды можно добавить любую программу Альфа-ДОС, например, другой, более привычный текстовый редактор, обучающую систему, часто используемые программы пользователя.

После выбора команды "Настроить меню" открывается кадр диалога (рис. 49).

Список команд:	
&Имя команды	<Добавить> <Изменить> <Удалить> <К началу> <К концу >
<Сохранить> <Снять> <Справка>	

Рис. 49. Кадр диалога команды "Настроить меню"

Для добавления новой команды в меню "Утилиты" надо выбрать командную кнопку <Добавить>. В результате откроется другой кадр диалога (рис. 50), в котором надо ввести имя новой команды, аргументы, необходимые для выполнения программы, полный маршрут поиска, каталог, который будет рассматриваться как текущий при выполнении программы. Кроме того, можно для этой новой команды определить сокращенный набор клавиш, чтобы выполнять эту команду, не открывая меню. Этот набор клавиш будет появляться рядом с именем команды в меню. При определении имени команды можно использовать символ '&' для выделения буквы, которая будет использоваться для выбора команды. Если этот знак не используется, то в качестве такой буквы будет использоваться первая буква команды.

Имя команды:	[	]
Маршрут:	[	]
Аргумент:	[	]
Каталог:	[	]
[X] Запрос до возврата		
Спецключ:	(.) Нет ()	АЛТ+Ф[]
< ОК > <Снять> <Справка>		

Рис. 50. Кадр диалога для добавления новой команды

16.7. СПРАВОЧНАЯ СИСТЕМА СРЕДЫ СИ

В интегрированной среде Си имеется мощная справочная система, позволяющая получить практически любую информацию о среде, которая может понадобиться пользователю. Команды меню "Справка" управляют получением справки во время работы в среде Си (рис. 51).

Существует несколько видов справки:

- справка о языке Си и функциях библиотеки;

Справка	
Индекс	
Общая	
Тема	Ф1
О справке	РЕГ+Ф1

- справка о сообщениях об ошибках;
- справка о меню среды Си и кадрах диалога.

Рис. 51. Меню "Справка"

Справочная система является контекстно чувствительной, т.е. использует положение курсора. Курсор определяет тему справки. Например, если курсор указывает на ключевое слово языка Си, то появится справка об этом ключевом слове. Если курсор находится в кадре диалога, то появится справка о данном кадре диалога.

Команда "О справке" (сокращенный набор клавиш РЕГ+Ф1) используется для получения общей информации о справочной системе среды Си и получении справки.

Команда "Общая" выводит таблицу содержимого справочной системы среды Си. Для перемещения по таблице надо использовать клавиши управления курсором. После выбора темы из таблицы для получения справки достаточно нажать клавишу Ф1.

Команда "Тема" (сокращенный набор клавиш Ф1) используется для справки по конкретной теме. Перед использованием этой команды курсор надо поместить на объект, о котором желательно получить справку.

Команда "Индекс" выводит алфавитный указатель тем. Используя этот указатель, можно выбрать букву, с которой начинается имя интересующего объекта. В результате будет выдан список элементов на эту букву. Выбрав из списка нужный элемент, надо нажать Ф1 для получения справки об этом объекте.

ЛИТЕРАТУРА

1. *Иванов А.Г.* Язык программирования Си: Предварительное описание// Прикладная информатика. — 1985 — Вып.1.— С.68-133.
2. *Керниган Б., Ритчи Д., Фьюэр А.* Язык программирования Си: Задачи по языку Си: Пер. с англ. — М.: Финансы и статистика, 1985.— 279с.
3. *Болски М.И.* Язык программирования Си: Пер. с англ. — М.: Радио и связь, 1988.— 96с.
4. *Берри Р., Микинз Б.* Язык Си: Введение для программистов: Пер с англ. — М.: Финансы и статистика, 1988.— 191с.
5. *Уэйт М., Прата С., Мартин Д.* Язык Си: Пер. с англ. — М.: Мир, 1988.— 512с.
6. *Дансмур М., Дейвис Г.* Операционная система UNIX и программирование на языке Си: Пер. с англ. — М.: Радио и связь, 1989.— 192с.
7. *Запольский А.П. и др.* Персональные компьютеры Единой системы ЭВМ/ А.П.Запольский, В.Я.Пыхтин, А.М.Чистяков, В.В.Шкляр. — М.: Финансы и статистика, 1988.— 143с.
8. *Брябрин В.М.* Программное обеспечение персональных ЭВМ.- М.: Наука, 1988.— 272с.

КЛАВИАТУРА ПЭВМ ЕС

Номера клавиш, скэн-коды и названия клавиш или соответствующие им символы для клавиатуры ПЭВМ ЕС1842 даны в табл. 1.

Таблица 1

Номер клавиши	Скэн-код	Название клавиши	Номер клавиши	Скэн-код	Название клавиши
1	01	ВЫХ	42	13	К R
2	3B	Ф1	43	14	Е Т
3	3C	Ф2	44	15	Н У
4	3D	Ф3	45	16	Г U
5	3E	Ф4	46	17	Ш I
6	3F	Ф5	47	18	Щ O
7	40	Ф6	48	19	З Р
8	41	Ф7	49	1A	Х
9	42	Ф8	50	1B	Ъ
10	43	Ф9	51	68	ИНФ
11	44	Ф10	52	Eo53	УД
12	57	Ф11	53	Eo4F	КОН
13	58	Ф12	54	Eo51	СТР↓
14	Eo2aEo37	ПЕЧ	55	47	7 НАЧ
15	46	␣ ЭКР	56	48	8 ↑
16	E11d45	СТОП	57	49	9 СТР↑
17	29	# ~ *	58	4A	-
18	02	+ (	59	3A	␣ ЗГЛ
19	03	1 &)	60	1E	Ф A
20	04	2 @ -	61	1F	Ы S
21	05	3 ^ /	62	20	В D
22	06	4 { "	63	21	А F
23	07	5 } :	64	22	П G
24	08	6 \ ,	65	23	Р H
25	09	7 < .	66	24	О J
26	0A	8 > _	67	25	Л K
27	0B	9 ?	68	26	Д L
28	0C	0 [%	69	27	Ж
29	0D	=] !	70	28	Э
30	2B	␣ ;	71	1C	ВВОД
31	DE	ВШ	73	4B	4 ←
32	Eo52	ВСТ	74	4C	5
33	Eo47	НАЧ	75	4D	6 →
34	Eo49	СТР↑	76	4E	+
35	45	␣ ЦИФ	77	2A	РЕГ
36	37	*	79	2C	Я Z
37	Eo05	/	80	2D	Ч X
38	OF	ТАБ	81	2E	С C
39	10	Й Q	82	2F	М V
40	11	Ц W	83	30	И B
41	12	У E	84	31	Т N

Продолжение табл. 1

Номер клавиши	Скэн-код	Название клавиши	Номер клавиши	Скэн-код	Название клавиши
85	32	Ь М	98	38	АЛТ
86	33	Б	99	39	ПРОБЕЛ
87	34	Ю	100	66	☐ РУС
88	35	Ё	101	67	Р/А
89	36	РЕГ	102	Eo4B	←
91	Eo48	↑	103	Eo50	↓
92	4F	1 КОН	104	Eo4D	→
93	50	2 ↓	105	52	0 ВСТ
94	51	3 СТР↓	107	53	. УД
96	65	Р/А	108	Eo1C	ВВОД
97	1D	УПР			

Соответствие названий клавиш на клавиатуре для различных моделей ПЭВМ ЕС и зарубежных ПЭВМ приведено в табл. 2.

Таблица 2

ЕС1842	ЕС1840 ЕС1841	Назначение клавиши	Зарубежная ПЭВМ
ЗГЛ	ФПБ	Фиксация прописных букв	Caps Lock
РУС	РУС	Фиксация алфавита	-
РЕГ	↑	Смена регистра	↑ Shift
Р/А	Р/Л	Смена регистра — алфавита	-
ТАБ	← --	Табуляция	Tab
	-- →		
ВВОД	ВВОД	Конец ввода	↵ Enter
УПР	УПР	Управляющие символы	Ctrl
АЛТ	ДОП	Альтернативный набор	Alt
ВШ	← --	Возврат на шаг	← -- Backspace
ВЫХ	КЛЮЧ	Выход	Esc
ИНФ	ИНФ	Вспомогательная информация	-
ПЕЧ	ПЕЧ / *	Печать экрана	Print Screen
ЭКР	ФСД/СТОП	Блокировка прокрутки	Scroll Lock
СТОП	ФСД/СТОП	Пауза	Pause/Break
Ф1...Ф10	Ф1...Ф10	Функциональные клавиши	F1...F10
Ф11,Ф12	-	Функциональные клавиши	F11,F12
ЦИФ	ЦИФ	Переключение на ввод цифр	Num Lock
ВСТ	ВСТ	Вставка символа	Ins
УД	УДЛ	Удаление символа	Del
НАЧ	↶	Переместить курсор в начало файла	Home
КОН	КОН	Переместить курсор в конец файла	End
СТР↑	↑ ■	Переместить курсор на страницу вверх	Page Up
СТР↓	↓ ■	Переместить курсор на страницу вниз	Page Down

ПРОТОТИПЫ ФУНКЦИЙ БИБЛИОТЕКИ СП СИ

1. Общие функции библиотеки

abort — Прерывает процесс и выдает код ошибки
#include `<process.h>` или `<stdlib.h>`
void abort(void);

```
abs, fabs, labs — Вычисляют абсолютное значение
#include <stdlib.h>
int abs(int x);
double fabs(double x);
long labs(long x);
```

access — Проверяет режим доступа к файлу

```
#include <io.h>
```

```
int access(char *path, int mode);
```

mode:	00 (существование)	04 (чтение разрешено)
	02 (запись разрешена)	06 (чтение & запись разрешены)

bdos — Активизирует системный вызов Альфа-ДОС

```
#include <dos.h>
```

```
int bdos(int dosfn, unsigned int dosdx, unsigned int dosal);
```

bessel — Вычисляет функцию Бесселя

```
#include <math.h>
```

```
double j0(double x);
```

```
double j1(double x);
```

```
double jn(int n, double x);
```

```
double y0(double x);
```

```
double y1(double x);
```

```
double yn(int n, double x);
```

_bios_disk — Вызывает дисковые функции (INT 0x13)

```
#include <bios.h>
```

```
unsigned _bios_disk(unsigned service, struct diskinfo_t *diskinfo);
```

```
service:  _DISK_RESET      _DISK_STATUS      _DISK_READ
           _DISK_WRITE     _DISK_VERIFY      _DISK_FORMAT
```

_bios_equiplist — Вызывает функции контроля оборудования (INT 0x11)

```
#include <bios.h>
```

```
unsigned _bios_equiplist(void);
```

_bios_keybrd — Вызывает функции для обслуживания клавиатуры (INT 0x16)

```
#include <bios.h>
```

```
unsigned _bios_keybrd(unsigned service);
```

```
service:  _KEYBRD_READ      _KEYBRD_READY      _KEYBRD_SHIFTSTATUS
           _NKEYBRD_READ    _NKEYBRD_READY    _NKEYBRD_SHIFTSTATUS
```

_bios_memsize — Вызывает функции для определения общего количества доступной памяти (INT 0x12)

```
#include <bios.h>
```

```
unsigned _bios_memsize(void);
```

_bios_printer — Вызывает функции для обслуживания вывода на печать (INT 0x17)

```
#include <bios.h>
```

```
unsigned _bios_printer(unsigned service, unsigned printer, unsigned databyte);
```

```
service:  _PRINTER_WRITE   _PRINTER_INIT   _PRINTER_STATUS
```

_bios_serialcom — Вызывает функции для управления коммуникационным адаптером (INT 0x14)

```
#include <bios.h>
```

```
unsigned _bios_serialcom(unsigned service, unsigned port, unsigned databyte);
```

```
service:  _COM_INIT      _COM_SEND      _COM_RECEIVE      _COM_STATUS
data:     _COM_CHR7     _COM_CHR8     _COM_STOP1      _COM_STOP2
           _COM_NOPARITY _COM_EVENPARITY _COM_ODDPARITY
           _COM_110     _COM_150      _COM_300        _COM_600
           _COM_1200    _COM_2400    _COM_4800       _COM_9600
```

(могут быть соединены знаком |)

_bios_timeofday — Вызывает функции для получения или установки текущего системного времени (INT 0x1A)

```
#include <bios.h>
unsigned _bios_timeofday(int service, long *timeval);
service: _TIME_GETCLOCK _TIME_SETCLOCK
```

bsearch — Выполняет двоичный поиск в отсортированном массиве

```
#include <stdlib.h>
#include <search.h>
void *bsearch(const void *key, const void *base, size_t num, size_t width,
              int (*compare)(const void *elem1, const void *elem2));
```

cabs — Вычисляет абсолютное значение комплексного числа

```
#include <math.h>
double cabs(struct complex z);
```

calloc — Размещает в памяти массив и инициализирует нулевыми элементами

```
#include <malloc.h>
#include <stdlib.h>
void *calloc(size_t n, size_t size);
```

ceil, floor — Вычисляют наименьшее (наибольшее) целое, большее (меньшее) или равное заданной величине

```
#include <math.h>
double ceil(double x);
double floor(double x);
```

cgets — Читает строку символов с консоли

```
#include <conio.h>
char *cgets(char *str);
```

chain_intr — Сцепляет обработчики прерываний

```
#include <dos.h>
void _chain_intr(void (interrupt far *target)());
```

chdir — Изменяет текущий рабочий каталог

```
#include <direct.h>
int chdir(char *path);
```

chmod — Изменяет установку режима доступа для файла

```
#include <io.h>
#include <sys\types.h>
#include <sys\stat.h>
int chmod(char *path, int pmode);
pmode: S_IWRITE S_IREAD S_IREAD | S_IWRITE
```

chsize — Изменяет размер файла

```
#include <io.h>
int chsize(int handle, long size);
```

_clear87, _status87 — Получают и очищают слово состояния плавающей точки

```
#include <float.h>
#include <math.h>
unsigned _clear87(void);
unsigned _status87(void);
```


clearerr — Очищает индикатор ошибки для потока

```
#include <stdio.h>
void clearerr(FILE *stream);
```

clock — Вычисляет время, использованное процессом

```
#include <time.h>
clock_t clock(void);
```

close — Закрывает файл

```
#include <io.h>
int close(int handle);
```

control87 — Получает и устанавливает управляющее слово плавающей точки

```
#include <float.h>
unsigned int _control87(unsigned int new, unsigned int mask);
```

cos, sin, tan — Вычисляют тригонометрические функции

```
#include <math.h>
double cos(double x);
double sin(double x);
double tan(double x);
```

cosh, sinh, tanh — Вычисляют гиперболические функции

```
#include <math.h>
double cosh(double x);
double sinh(double x);
double tanh(double x);
```

cprintf — Форматирует данные и выводит их на консоль

```
#include <conio.h>
int cprintf(char *format[, argument]...);
```

cputs — Выводит строку символов на консоль

```
#include <conio.h>
int cputs(char *string);
```

creat — Создает новый файл

```
#include <io.h>
#include <sys\types.h>
#include <sys\stat.h>
int creat(char *path, int pmode);
pmode: S_IWRITE S_IREAD S_IREAD | S_IWRITE
```

cscanf — Читает форматизованные данные с консоли

```
#include <conio.h>
int cscanf(char *format[, argument]...);
```

ctime — Преобразует время `time_t` в строку символов

```
#include <time.h>
char *ctime(const time_t *time);
```

dieetomsbin, dmsbintoieee, fieetomsbin, fmsbintoieee — Преобразуют число в формате IEEE в двоичный формат С П Си

```
#include <float.h>
#include <math.h>
int dieetomsbin(double *src8, double *dst8);
int dmsbintoieee(double *src8, double *dst8);
int fieetomsbin(float *src4, float *dst4);
int fmsbintoieee(float *src4, float *dst4);
```


difftime — Вычисляет разницу между двумя значениями времени

```
#include <time.h>
```

```
double difftime(time_t time2, time_t time1);
```

_disable, _enable — Запрещает (разрешает) прерывания

```
#include <dos.h>
```

```
void _disable(void);
```

```
void _enable(void);
```

div, ldiv — Вычисляет частное и остаток двух целых величин

```
#include <stdlib.h>
```

```
div_t div(int numer, int denom);
```

```
ldiv_t ldiv(long int numer, long int denom);
```

_dos_allocmem — Распределяет блок памяти (используя системный вызов Альфа-ДОС 0x48)

```
#include <dos.h>
```

```
unsigned _dos_allocmem(unsigned size, unsigned *segment);
```

_dos_close — Закрывает файл (INT 0x3E)

```
#include <dos.h>
```

```
unsigned _dos_close(int handle);
```

_dos_creat, _dos_creatnew — Создают файл (INT 0x3C и 0x5B)

```
#include <dos.h>
```

```
unsigned _dos_creat(char *path, unsigned attribute, int *handle);
```

```
unsigned _dos_creatnew(char *path, unsigned attribute, int *handle);
```

_dos_findfirst, _dos_findnext — Ищут файл с определенными атрибутами (INT 0x4E)

```
#include <dos.h>
```

```
unsigned _dos_findfirst(char *path, unsigned attribute, struct find_t *buffer);
```

```
unsigned _dos_findnext(struct find_t *buffer);
```

attribute: **_A_NORMAL** **_A_RDONLY** **_A_HIDDEN** **_A_SYSTEM**
_A_VOLID **_A_SUBDIR** **_A_ARCH**
(могут быть соединены знаком |)

_dos_freemem — Освобождает блок памяти (INT 0x49)

```
#include <dos.h>
```

```
unsigned _dos_freemem(unsigned segment);
```

_dos_getdate — Получает текущую системную дату (INT 0x2A)

```
#include <dos.h>
```

```
void _dos_getdate(struct dosdate_t *date);
```

_dos_getdiskfree — Получает информацию о диске (INT 0x36)

```
#include <dos.h>
```

```
unsigned _dos_getdiskfree(unsigned drive, struct diskfree_t *diskinfo);
```

_dos_getdrive — Получает имя текущего дисковода (INT 0x19)

```
#include <dos.h>
```

```
void _dos_getdrive(unsigned *drive);
```


_dos_getfileattr — Получает текущие атрибуты файла или каталога (INT 0x43)

```
#include <dos.h>
```

```
unsigned _dos_getfileattr(char *path, unsigned *attributes);
```

_dos_getftime — Получает дату и время последней записи в файл (INT 0x57)

```
#include <dos.h>
```

```
unsigned _dos_getftime(int handle, unsigned *date, unsigned *time);
```

_dos_gettime — Получает текущее системное время (INT 0x2C)

```
#include <dos.h>
```

```
void _dos_gettime(struct dostime_t *time);
```

_dos_getvect, _dos_setvect — Получает и устанавливает текущее значение вектора прерывания (INT 0x35, INT 0x25)

```
#include <dos.h>
```

```
void (interrupt far *_dos_getvect (unsigned intnum)) ();
```

```
void _dos_setvect(unsigned intnum, void (interrupt far *inthandler) ());
```

_dos_keep — Устанавливает резидентную программу в памяти (INT 0x31)

```
#include <dos.h>
```

```
void _dos_keep(unsigned retcode, unsigned memsize);
```

_dos_open — Открывает файл (INT 0x3D)

```
#include <dos.h>
```

```
#include <fcntl.h>
```

```
#include <share.h>
```

```
unsigned _dos_open(char *path, unsigned mode, int *handle);
```

```
mode: O_RDONLY    O_WRONLY    O_RDWR  SH_COMPAT  SH_DENYRW
      SH_DENYWR   SH_DENYRD   SH_DENYNONE  O_NOINHERIT
      (могут быть соединены знаком | )
```

_dos_read — Читает данные из файла (INT 0x3F)

```
#include <dos.h>
```

```
unsigned _dos_read(int handle, void far *buffer,
    unsigned count, unsigned *bytes);
```

_dos_setblock — Изменяет размер сегмента (INT 0x4A)

```
#include <dos.h>
```

```
unsigned _dos_setblock(unsigned size, unsigned segment, unsigned *maxsize);
```

_dos_setdate — Устанавливает текущую системную дату (INT 0x2B)

```
#include <dos.h>
```

```
unsigned _dos_setdate(struct dosdate_t *date);
```

_dos_setdrive — Устанавливает дисковод по умолчанию (INT 0x0E)

```
#include <dos.h>
```

```
void _dos_setdrive(unsigned drivenum, unsigned *drives);
```

_dos_setfileattr — Устанавливает атрибуты файла или каталога (INT 0x43)

```
#include <dos.h>
```

```
unsigned _dos_setfileattr(char *path, unsigned attribute);
```

```
attribute: _A_NORMAL  _A_RDONLY  _A_HIDDEN  _A_SYSTEM
           _A_VOLID   _A_SUBDIR  _A_ARCH
           (могут быть соединены знаком | )
```


_dos_setftime — Устанавливает дату и время последней записи в файл (INT 0x57)

```
#include <dos.h>
unsigned _dos_setftime(int handle, unsigned date,
                      unsigned time);
```

_dos_settime — Устанавливает текущее системное время (INT 0x2D)

```
#include <dos.h>
unsigned _dos_settime(struct dostime_t *time);
```

_dos_write — Записывает буфер в файл (INT 0x40)

```
#include <dos.h>
unsigned _dos_write(int handle, void far *buffer,
                  unsigned count, unsigned *bytes);
```

dosexterr — Получает значения регистров из системного вызова INT 0x59

```
#include <dos.h>
int dosexterr(struct DOSERROR *buffer);
```

dup, dup2 — Создает второй дескриптор для открытого файла (переназначает дескриптор файла)

```
#include <io.h>
int dup(int handle);
int dup2(int handle1, int handle2);
```

ecvt, fcvt, gcvt — Преобразуют число с плавающей точкой в строку символов

```
#include <stdlib.h>
char *ecvt(double value, int count, int *dec, int *sign);
char *fcvt(double value, int count, int *dec, int *sign);
char *gcvt(double value, int digits, char *buffer);
```

eof — Анализирует условие конца файла

```
#include <io.h>
int eof(int handle);
```

exec, execl, execl, execlp, execlpe, execv, execve, execvp, execvpe — Выполняют порожденный процесс

```
#include <process.h>
int execl(char *path, char *arg0,
          char *arg1, ..., char *argn, NULL);
int execl(char *path, char *arg0,
          char *arg1, ..., char *argn, NULL, char *envp[ ]);
int execlp(char *path, char *arg0,
          char *arg1, ..., char *argn, NULL);
int execlpe(char *path, char *arg0,
          char *arg1, ..., char *argn, NULL, char *envp[ ]);
int execv(char *path, char *argv[ ]);
int execve(char *path, char *argv[ ], char *envp[ ]);
int execvp(char *path, char *argv[ ]);
int execvpe(char *path, char *argv[ ], char *envp[ ]);
```

exit, _exit — Завершают процесс после очистки

```
#include <process.h> или <stdlib.h>
void exit(int status);
void _exit(int status);
```


exp — Вычисляет экспоненциальную функцию
#include <math.h>
double exp(double x);

_expand — Изменяет размер блока памяти
#include <malloc.h>
void *_expand(void *block, size_t size);

fclose, fcloseall — Закрывают поток
#include <stdio.h>
int fclose(FILE *stream);
int fcloseall(void);

fdopen — Открывает поток, используя дескриптор
#include <stdio.h>
FILE *fdopen(int handle, char *type);
type: r w a r+ w+ a+
t b (присоединяется к type для указания режима)

feof — Анализирует условие конца файла для потока
#include <stdio.h>
int feof(FILE *stream);

ferror — Анализирует поток на ошибки
#include <stdio.h>
int ferror(FILE *stream);

fflush — Освобождает поток
#include <stdio.h>
int fflush(FILE *stream);

fgetc, fgetchar — Читают символ из потока
#include <stdio.h>
int fgetc(FILE *stream);
int fgetchar(void);

fgetpos, fsetpos — Получают указатель позиции в потоке
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, const fpos_t *pos);

fgets — Получает строку из потока
#include <stdio.h>
char *fgets(char *string, int n, FILE *stream);

filelength — Получает длину файла
#include <io.h>
long filelength(int handle);

fileno — Получает дескриптор файла, связанный с потоком
#include <stdio.h>
int fileno(FILE *stream);

flushall — Освобождает потоки, выгружает все буферы
#include <stdio.h>
int flushall(void);

fmod — Вычисляет остаток с плавающей точкой

```
#include <math.h>
double fmod(double x, double y);
```

fopen — Открывает файл

```
#include <stdio.h>
FILE *fopen(const char *path, const char *type);
type: r w a r+ w+ a+
      t b (присоединяется к type для указания режима)
```

FP_OFF, FP_SEG — Получают или устанавливают компонент смещения (сегмента) указателя far

```
#include <dos.h>
unsigned FP_OFF(void far *address);
unsigned FP_SEG(void far *address);
```

_fpret — Повторно инициализирует пакет математических функций с плавающей точкой

```
#include <float.h>
void _fpret(void);
```

fprintf — Записывает форматизованные данные в поток

```
#include <stdio.h>
int fprintf(FILE *stream, const char *format[, argument] ...);
```

fputc, fputchar — Записывает символ в поток

```
#include <stdio.h>
int fputc(int c, FILE *stream);
int fputchar(int c);
```

fputs — Записывает строку в поток

```
#include <stdio.h>
int fputs(char *string, FILE *stream);
```

fread — Читает данные из потока

```
#include <stdio.h>
size_t fread(void *buffer, size_t size,
             size_t count, FILE *stream);
```

free, _ffree, _nfree — Освобождают блок памяти

```
#include <malloc.h>
#include <stdlib.h> (для совместимости с АНИС только free)
void _ffree(void far *buffer);
void free(void *buffer);
void _nfree(void near *buffer);
```

_freect — Возвращает количество памяти, доступное для распределения

```
#include <malloc.h>
unsigned int _freect(size_t size);
```

freopen — Переназначает указатель файла

```
#include <stdio.h>
FILE *freopen(const char *path, const char *type,
              FILE *stream);
type: r w a r+ w+ a+
      t b (присоединяется к type для указания режима)
```


frexp — Получает мантиссу и экспоненту числа с плавающей точкой

#include <math.h>

double frexp(double x, int *exp_ptr);

fscanf — Читает форматизованные данные из потока

#include <stdio.h>

int fscanf(FILE *stream, const char *format [, argument] ...);

fseek — Переустанавливает указатель файла на указанную позицию

#include <stdio.h>

int fseek(FILE *stream, long offset, int origin);

origin: SEEK_SET, SEEK_CUR, SEEK_END

fstat, stat — Получают информацию о состоянии файла

#include <sys/types.h>

#include <sys/stat.h>

int fstat(int handle, struct stat *buffer);

int stat(char *path, struct stat *buffer);

ftell — Получает указатель текущей позиции файла

#include <stdio.h>

long ftell(FILE *stream);

ftime — Получает текущее системное время

#include <sys/types.h>

#include <sys/timeb.h>

void ftime(struct timeb *timeptr);

fwrite — Записывает данные в поток

#include <stdio.h>

size_t fwrite(const void *buffer, size_t size, size_t count, FILE *stream);

getc, getchar — Читают символ из потока

#include <stdio.h>

int getc(FILE *stream);

int getchar(void);

getch, getche — Читают символ с терминала без отображения и с отображением

#include <conio.h>

int getch(void);

int getche(void);

getcwd — Получает текущий рабочий каталог

#include <direct.h>

char *getcwd(char *path, int n);

getenv — Получает значение переменной среды

#include <stdlib.h>

char *getenv(const char *varname);

getpid — Получает идентифицирующий номер процесса

#include <process.h>

int getpid(void);

gets — Читает строку из потока stdin

#include <stdio.h>

char *gets(char *buffer);

getw — Читает данное типа `int` из потока

```
#include <stdio.h>
int getw(FILE *stream);
```

gmtime — Преобразует время в структуру

```
#include <time.h>
struct tm *gmtime(const time_t *time);
```

halloc — Распределяет блок памяти `huge`

```
#include <malloc.h>
void huge *halloc(long n, size_t size);
```

_harderr, _hardresume, _hardretn — Устанавливают обработчик постоянной ошибки для `INT 0x24`, возвращают управление из программы обработки прерываний в систему

```
#include <dos.h>
void _harderr(void (far *fptr)());
void _hardresume(int result);
result:    _HARDERR_IGNORE    _HARDERR_RETRY
           _HARDERR_ABORT    _HARDERR_FAIL
void _hardretn(int error);
```

_heapchk, _fheapchk, _nheapchk — Проверяют правильность кучи

```
#include <malloc.h>
int _heapchk(void);
int _fheapchk(void);
int _nheapchk(void);
```

_heapset, _fheapset, _nheapset — Заполняют кучу заданным значением

```
#include <malloc.h>
int _heapset(unsigned fill);
int _fheapset(unsigned fill);
int _nheapset(unsigned fill);
```

_heapwalk, _fheapwalk, _nheapwalk — Тестируют кучу

```
#include <malloc.h>
int _heapwalk(struct _heapinfo *entry);
int _fheapwalk(struct _heapinfo *farentry);
int _nheapwalk(struct _heapinfo *nearentry);
```

hfree — Освобождает блок памяти `huge`

```
#include <malloc.h>
void hfree(void huge *buffer);
```

hypot — Вычисляет длину гипотенузы

```
#include <math.h>
double hypot(double x, double y);
```

inp, inpw, outp, outpw — Выполняют ввод-вывод для порта

```
#include <conio.h>
int inp(unsigned port);
unsigned inpw(unsigned port);
int outp(unsigned port, int byte);
unsigned outpw(unsigned port, unsigned word);
```


int86 — Выполняет прерывание Альфа-ДОС

#include <dos.h>

int int86 (int intno, union REGS *inregs, union REGS *outregs);

int86x — Выполняет прерывание Альфа-ДОС со значениями регистра сегмента

#include <dos.h>

int int86x (int intno, union REGS *inregs,
union REGS *outregs, struct SREGS *segregs);

intdos — Выполняет системный вызов Альфа-ДОС

#include <dos.h>

int intdos (union REGS *inregs, union REGS *outregs);

intdosx — Выполняет системный вызов Альфа-ДОС со значениями регистра сегмента

#include <dos.h>

int intdosx (union REGS *inregs, union REGS *outregs,
struct SREGS *segregs);

isalnum, isalpha, isascii, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit — Выполняют проверку и преобразование символов

#include <ctype.h>

int isalnum (int c); (буквенно-цифровой символ)

int isalpha (int c); (буквенный символ)

int isascii (int c); (символ ASCII)

int iscntrl (int c); (управляющий символ)

int isdigit (int c); (десятичная цифра)

int isgraph (int c); (печатаемый символ, не пробел)

int islower (int c); (строчная буква)

int isprint (int c); (печатаемый символ)

int ispunct (int c); (символ пунктуации)

int isspace (int c); (пробельный символ)

int isupper (int c); (прописная буква)

int isxdigit (int c); (шестнадцатеричная цифра)

isatty — Проверяет на символьное устройство

#include <io.h>

int isatty (int handle);

itoa, ltoa, ultoa — Преобразуют целое в строку символов

#include <stdlib.h>

char *itoa (int value, char *string, int radix);

char *ltoa (long value, char *string, int radix);

char *ultoa (unsigned long value, char *string, int radix);

kbhit — Проверяет терминал на нажатие клавиши

#include <conio.h>

int kbhit (void);

ldexp — Преобразует мантиссу и экспоненту в значение с плавающей точкой

#include <math.h>

double ldexp (double x, int exp);

lfind, lsearch — Линейный поиск заданного значения ключа

```
#include <stdlib.h>
#include <search.h>
char *lfind(char *key, char *base, unsigned *num,
            unsigned width, int (*compare)
            (const void *elem1, const void *elem2));
char *lsearch(char *key, char *base, unsigned *num,
            unsigned width, int (*compare)
            (const void *elem1, const void *elem2));
```

localtime — Преобразует время из типа int в структуру с корректировкой в местное

```
#include <time.h>
struct tm *localtime(const time_t *time);
```

locking — Блокирует и разблокирует байты файла

```
#include <io.h>
#include <sys\locking.h>
int locking(int handle, int mode, long nbyte);
mode:    LK_LOCK LK_RLCK LK_NBLCK LK_NBRLCK LK_UNLCK
```

log, log10 — Вычисляют натуральный (десятичный) логарифм

```
#include <math.h>
double log(double x);
double log10(double x);
```

longjmp — Восстанавливает состояние программы

```
#include <setjmp.h>
void longjmp(jmp_buf env, int value);
```

_lrotl, _lrotr, _rotl, _rotr — Выполняют сдвиги вправо и влево

```
#include <stdlib.h>
unsigned long _lrotl(unsigned long value, int shift);
unsigned long _lrotr(unsigned long value, int shift);
unsigned _rotl(unsigned int value, int shift);
unsigned _rotr(unsigned int value, int shift);
```

lseek — Переустанавливает указатель файла

```
#include <io.h>
#include <stdio.h>
long lseek(int handle, long offset, int origin);
origin:    SEEK_SET    SEEK_CUR    SEEK_END
```

_makepath — Создает единственное имя маршрута

```
#include <stdlib.h>
void _makepath(char *path, char *drive, char *dir,
               char *fname, char *ext);
```

malloc, _fmalloc, _nmalloc — Распределяют блок памяти

```
#include <malloc.h>
#include <stdlib.h> (для совместимости с АНИС только malloc)
void *malloc(size_t size);
void far *_fmalloc(size_t size);
void near *_nmalloc(size_t size);
```

matherr — Обрабатывает ошибки, генерируемые математическими функциями

```
#include <math.h>
int matherr(struct exception *except);
```


max, min — Возвращают большее (меньшее) значение

#include <stdlib.h>

type max(type a, type b); /* Макроопределение */

type min(type a, type b); /* Макроопределение */

_memavl — Возвращает размер доступной памяти

#include <malloc.h>

size_t _memavl(void);

memccpy — Копирует символы из буфера

#include <string.h> или <memory.h>

void *memccpy(void *dest, void *src, int c, unsigned count);

memchr — Ищет заданный символ в буфере

#include <string.h> или <memory.h>

void *memchr(const void *buf, int c, size_t count);

memcmp, memicmp — Сравнивают заданные символы двух буферов

#include <string.h> или <memory.h>

int memcmp(const void *buf1, const void *buf2, size_t count);

int memicmp(void *buf1, void *buf2, unsigned int count);

memcpy — Копирует символы из одного буфера в другой

#include <string.h> или <memory.h>

void *memcpy(void *dest, const void *src, size_t count);

_memmax — Ищет размер наибольшего непрерывного блока памяти

#include <malloc.h>

#include <stdlib.h>

size_t _memmax(void);

memmove — Копирует символы из одного буфера в другой

#include <string.h>

void *memmove(void *dest, const void *src, size_t count);

memset — Инициализирует буфер определенным символом

#include <string.h> или <memory.h>

void *memset(void *dest, int c, size_t count);

mkdir — Создает новый каталог

#include <direct.h>

int mkdir(char *path);

mktemp — Создает уникальное имя файла

#include <io.h>

char *mktemp(char *template);

mktime — Преобразует местное время в календарное значение

#include <time.h>

time_t mktime(struct tm *timeptr);

modf — Разбивает значение с плавающей точкой на целую и дробную части

#include <math.h>

double modf(double x, double *intptr);

movedata — Копирует символы в другой сегмент

#include <string.h>

void movedata(unsigned srcseg, unsigned srcoff,
 unsigned destseg, unsigned destoff, unsigned nbytes);

_msize, _fmsize, _nmsize — Возвращают размер блока памяти

```
#include <malloc.h>
size_t _msize(void *buffer);
size_t _fmsize(void far *buffer);
size_t _nmsize(void near *buffer);
```

open — Открывает файл

```
#include <fcntl.h>
#include <io.h>
#include <sys\types.h>
#include <sys\stat.h>
int open(char *path, int flag[, int mode]);
flag:  O_APPEND  O_BINARY  O_CREAT  O_EXCL  O_RDONLY
       O_RDWR   O_TEXT    O_TRUNC  O_WRONLY
      (могут быть соединены знаком | )
mode:  S_IWRITE  S_IREAD   S_IREAD | S_IWRITE
```

perror — Печатает сообщение об ошибке

```
#include <stdio.h>
void perror(const char *string);
int errno;
int sys_nerr;
char *sys_errlist[sys_nerr];
```

pow — Вычисляет значение x в степени y

```
#include <math.h>
double pow(double x, double y);
```

printf — Записывает форматизованные данные в stdout

```
#include <stdio.h>
int printf(const char *format[, argument]...);
```

putc, putchar — Записывает символ в поток

```
#include <stdio.h>
int putc(int c, FILE *stream);
int putchar(int c);
```

putch — Записывает символ на консоль

```
#include <conio.h>
int putch(int c);
```

putenv — Создает новые переменные среды

```
#include <stdlib.h>
int putenv(char *envstring);
```

puts — Записывает строку в stdout

```
#include <stdio.h>
int puts(const char *string);
```

putw — Записывает целое в поток

```
#include <stdio.h>
int putw(int binint, FILE *stream);
```


qsort — Выполняет быструю сортировку

```
#include <stdlib.h>
```

```
#include <search.h>
```

```
void qsort(void *base, size_t num, size_t width,  
           int (*compare)(const void *elem1, const void *elem2));
```

raise — Посылает сигнал выполняющейся программе

```
#include <signal.h>
```

```
int raise(int sig);
```

```
sig:  SIGABRT  SIGILL  SIGSEGV  
      SIGFPE  SIGINT  SIGTERM
```

rand, srand — Генерируют псевдослучайное число

```
#include <stdlib.h>
```

```
int rand(void);
```

```
void srand(unsigned seed);
```

read — Читает данные из файла

```
#include <io.h>
```

```
int read(int handle, char *buffer, unsigned count);
```

realloc — Перераспределяет блок памяти

```
#include <malloc.h>
```

```
#include <stdlib.h>
```

```
void *realloc(void *buffer, size_t size);
```

remove, unlink — Удаляют файл

```
#include <io.h> или <stdio.h>
```

```
int remove(const char *path);
```

```
int unlink(const char *path);
```

rename — Переименовывает файл или каталог

```
#include <io.h> или <stdio.h>
```

```
int rename(const char *oldname, const char *newname);
```

rewind — Переустанавливает указатель файла на начало файла

```
#include <stdio.h>
```

```
void rewind(FILE *stream);
```

rmdir — Удаляет каталог

```
#include <direct.h>
```

```
int rmdir(char *path);
```

rmtmp — Удаляет временные файлы

```
#include <stdio.h>
```

```
int rmtmp(void);
```

sbrk — Переустанавливает значение разрыва для вызывающего процесса

```
#include <malloc.h>
```

```
void *sbrk(int incr);
```

scanf — Читает форматизованные данные из stdin

```
#include <stdio.h>
```

```
int scanf(const char *format[, argument] ...);
```

_searchenv — Ищет файл по маршрутам, определенным в переменной среды

```
#include <stdlib.h>
```

```
char *_searchenv(char *name, char *env_var, char *path);
```


segread — Получает текущие значения регистров сегмента

```
#include <dos.h>
```

```
void segread (struct SREGS *segregs);
```

setbuf — Управляет буферизацией потока

```
#include <stdio.h>
```

```
void setbuf(FILE *stream, char *buffer);
```

setjmp — Сохраняет состояние программы

```
#include <setjmp.h>
```

```
int setjmp(jmp_buf env);
```

setmode — Устанавливает режим преобразования файла

```
#include <fcntl.h>
```

```
#include <io.h>
```

```
int setmode(int handle, int mode);
```

```
mode: O_TEXT O_BINARY
```

setvbuf — Управляет буферизацией потока и размером буфера

```
#include <stdio.h>
```

```
int setvbuf(FILE *stream, char *buffer, int type, size_t size);
```

```
type: _IOFBF _IOLBF _IONBF
```

signal — Устанавливает обработчик сигнала прерываний

```
#include <signal.h>
```

```
void (*signal(int sig, void (*func)(int sig, [, subcode])))(int sig);
```

```
sig: SIGABRT SIGILL SIGSEGV SIGFPE SIGINT SIGTERM
```

sopen — Открывает файл для совместного использования

```
#include <fcntl.h>
```

```
#include <io.h>
```

```
#include <share.h>
```

```
#include <sys\types.h>
```

```
#include <sys\stat.h>
```

```
int sopen(char *path, int oflag, int shflag[, int pmode]);
```

```
flag: O_APPEND O_BINARY O_CREAT O_EXCL O_RDONLY
      O_RDWR O_TEXT O_TRUNC O_WRONLY
```

(могут быть соединены знаком |)

```
shflag: SH_COMPAT SH_DENYRW SH_DENYWR
        SH_DENYRD SH_DENYNO
```

```
mode: S_IWRITE S_IREAD S_IREAD | S_IWRITE
```

spawnl, spawnle, spawnlp, spawnlpe, spawnv, spawnve, spawnvp, spawnvpe — Выполняют порожденный процесс

```
#include <process.h>
```

```
#include <stdio.h>
```

```
int spawnl(int modeflag, char *path, char *arg0,
           char *arg1,...char *argn, NULL);
```

```
int spawnle(int modeflag, char *path, char *arg0,
           char *arg1,...char *argn, NULL, char *envp[ ]);
```

```
int spawnlp(int modeflag, char *path, char *arg0,
           char *arg1,...char *argn, NULL);
```



```

int spawnlpe(int modeflag, char *path, char *arg0,
             char *arg1,...char *argn, NULL, char *envp[ ]);
int spawnv(int modeflag, char *path, char *argv[ ]);
int spawnve(int modeflag, char *path, char *argv[ ], char *envp[ ]);
int spawnvp(int modeflag, char *path, char *argv[ ]);
int spawnvpe(int modeflag, char *path, char *argv[ ], char *envp[ ]);
modeflag:   P_WAIT   P_NOWAIT   P_OVERLAY

```

_splitpath — Разбивает имя маршрута на составные компоненты

```
#include <stdlib.h>
```

```
void _splitpath(char *path, char *drive, char *dir, char *fname, char *ext);
```

sprintf — Записывает форматизованные данные в строку

```
#include <stdio.h>
```

```
int sprintf(char *buffer, const char *format[,argument] ...);
```

sqrt — Вычисляет квадратный корень

```
#include <math.h>
```

```
double sqrt(double x);
```

sscanf — Читает форматизованные данные из строки

```
#include <stdio.h>
```

```
int sscanf(const char *buffer, const char * format[,argument]);
```

stackavail — Получает размер доступной области стека

```
#include <malloc.h>
```

```
size_t stackavail(void);
```

strcat, strncat — Сцепляют строки

```
#include <string.h>
```

```
char *strcat(char * dest, const char * src);
```

```
char *strncat(char * dest, const char * src, size_t n);
```

strchr, strrchr, strstr — Находят символ в строке

```
#include <string.h>
```

```
char *strchr(const char *string, int c);
```

```
char *strrchr(const char *string, int c);
```

```
char *strstr(const char *string1, const char *string2);
```

strcmp, strcmpi, stricmp, strncmp, strnicmp — Сравнивают строки

```
#include <string.h>
```

```
int strcmp(const char *string1, const char *string2);
```

```
int strcmpi(const char *string1, const char *string2);
```

```
int stricmp(const char *string1, const char *string2);
```

```
int strncmp(const char *string1, const char *string2, size_t n);
```

```
int strnicmp(const char *string1, const char *string2, size_t n);
```


strcpy, strdup, strncpy — Копируют строку

```
#include <string.h>
char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
char *strdup(const char *string);
```

strcspn, strspn — Ищет подстроку в строке

```
#include <string.h>
size_t strcspn(const char *string1, const char *string2);
size_t strspn(const char *string1, const char *string2);
```

strnset, strset — Инициализируют символы строки

```
#include <string.h>
char *strnset(char *string, int c, size_t n);
char *strset(char *string, int c);
```

_strdate, _strtime — Копируют дату в буфер

```
#include <time.h>
char *_strdate(char *date);
char *_strtime(char *time);
```

strerror, _strerror — Получают системное сообщение об ошибке

```
#include <string.h>
char *strerror(int errnum);
char *_strerror(char *string);
int errno;
int sys_nerr;
char *sys_errlist[sys_nerr];
```

strlen — Получает длину строки

```
#include <string.h>
size_t strlen(const char *string);
```

strlwr,strupr — Преобразуют символы строки в строчные буквы и обратно

```
#include <string.h>
char *strlwr(char *string);
char *strupr(char *string);
```

strpbrk — Ищет в строке символ из заданного множества символов

```
#include <string.h>
char *strpbrk(const char *string1, const char *string2);
```

strrev — Переставляет в обратном порядке символы строки

```
#include <string.h>
char *strrev(char *string);
```

strtod, strtol, strtoul — Преобразуют строку в величину double, long и unsigned long

```
#include <stdlib.h>
double strtod(const char *nptr, char **endptr);
long strtol(const char *nptr, char **endptr, int base);
unsigned long strtoul(const char *nptr, char **endptr, int base);
```

strtok — Ищет следующую лексему в строке

```
#include <string.h>
char *strtok(char *string1, const char *string2);
```


swab — Меняет местами смежную пару байтов данных

#include <stdlib.h>

void swab(char *source, char *destination, int n);

system — Выполняет команду Альфа-ДОС

#include <process.h>

#include <stdlib.h>

int system(const char *string);

tell — Получает текущую позицию указателя файла

#include <io.h>

long tell(int handle);

tempnam, tmpnam — Создают временный файл в другом каталоге

#include <stdio.h>

char *tempnam(char *dir, char *prefix);

char *tmpnam(char *string);

time — Получает системное время

#include <time.h>

time_t time(time_t *timeptr);

tmpfile — Создает временный файл

#include <stdio.h>

FILE *tmpfile(void);

toascii, tolower, _tolower, toupper, _toupper — Функции преобразования символов

#include <ctype.h>

int toascii(int c); /* в символ ASCII */

int tolower(int c); /* в строчной символ */

int _tolower(int c); /* в строчной символ */

int toupper(int c); /* в прописной символ */

int _toupper(int c); /* в прописной символ */

tzset — Устанавливает переменные среды для времени

#include <time.h>

void tzset(void);

int daylight;

long timezone;

char *tzname[2];

umask — Устанавливает маску режима доступа по умолчанию

#include <io.h>

#include <sys\types.h>

#include <sys\stat.h>

int umask(int pmode);

pmode: S_IWRITE S_IREAD S_IWRITE | S_IREAD

ungetc — Возвращает символ обратно в поток

#include <stdio.h>

int ungetc(int c, FILE *stream);

ungetch — Возвращает обратно последний прочитанный с консоли символ

#include <conio.h>

int ungetch(int c);

utime — Устанавливает время модификации файла

```
#include <sys/types.h>
#include <sys/utime.h>
int utime(char *path, struct utimbuf *times);
```

va_arg, va_end, va_start — Обеспечивают доступ к переменному числу аргументов

```
#include <stdarg.h>
#include <stdio.h>
type va_arg(va_list arg_ptr, type);
void va_end(va_list arg_ptr);
void va_start(va_list arg_ptr, prev_param);
```

vfprintf, vprintf, vsprintf — Записывают форматизованные данные в поток, stdout и строку

```
#include <stdio.h>
#include <stdarg.h>
int vfprintf(FILE *stream, const char *format, va_list argptr);
int vprintf(const char *format, va_list argptr);
int vsprintf(char *buffer, const char *format, va_list argptr);
```

write — Записывает данные в файл

```
#include <io.h>
int write(int handle, char *buffer, unsigned count);
```

2. Графические функции

_arc, arc_w, _arc_wxy — Вычерчивают дугу

```
#include <graph.h>
short far _arc(short x1, short y1, short x2, short y2,
               short x3, short y3, short x4, short y4);
short far _arc_w(double wx1, double wy1,
                 double wx2, double wy2, double wx3, double wy3,
                 double wx4, double wy4);
short far _arc_wxy(struct _wxycoord *pwxyl,
                  struct _wxycoord *pwxyl2, struct _wxycoord *pwxyl3,
                  struct _wxycoord *pwxyl4);
```

_clearscreen — Чистит определенную область экрана

```
#include <graph.h>
void far _clearscreen(short area);
area: _GCLEARSCREEN _GVIEWPORT _GWINDOW
```

_displaycursor — Устанавливает переключатель курсора для графических функций

```
#include <graph.h>
short far _displaycursor(short toggle);
toggle: _G_CURSORON _G_CURSOROFF
```

_ellipse, _ellipse_w, _ellipse_wxy — Вычерчивают эллипс

```
#include <graph.h>
short far _ellipse(short control, short x1, short y1, short x2, short y2);
short far _ellipse_w(short control, double wx1, double wy1,
                    double wx2, double wy2);
short far _ellipse_wxy(short control, struct _wxycoord *pwxyl,
                      struct _wxycoord *pwxyl2);
control: _GFILLINTERIOR _GBORDER
```


_floodfill, _floodfill_w — Закрашивают область вывода

```
#include <graph.h>
```

```
short far _floodfill(short x, short y, short boundary);
```

```
short far _floodfill_w(double wx, double wy, short boundary);
```

_getactivepage — Получает номер текущей активной страницы

```
#include <graph.h>
```

```
short far _getactivepage(void);
```

_getbkcolor — Получает текущий цвет фона

```
#include <graph.h>
```

```
long far _getbkcolor(void);
```

_getcolor — Получает текущий цвет

```
#include <graph.h>
```

```
short far _getcolor(void);
```

_getcurrentposition, _getcurrentposition_w — Получают относительные или масштабированные координаты точки

```
#include <graph.h>
```

```
struct xycoord far _getcurrentposition(void);
```

```
struct _wxycoord far _getcurrentposition_w(void);
```

_getfillmask — Получает текущую маску закрашивания

```
#include <graph.h>
```

```
unsigned char far * far _getfillmask(unsigned char far * mask);
```

_getfontinfo — Получает характеристики текущего шрифта

```
#include <graph.h>
```

```
short far _getfontinfo(struct _fontinfo far * fontinfo);
```

_getgttextent — Получает ширину в пикселях текста, использующего шрифт

```
#include <graph.h>
```

```
short far _getgttextent(unsigned char far * text);
```

_getimage, _getimage_w, _getimage_wxy — Сохраняют изображение экрана в буфере памяти

```
#include <graph.h>
```

```
void far _getimage(short x1, short y1, short x2, short y2,
```

```
char far *image);
```

```
void far _getimage_w(double wx1, double wy1,
```

```
double wx2, double wy2, char far *image);
```

```
void far _getimage_wxy(struct _wxycoord * pwxyl,
```

```
struct _wxycoord * pwxyl2, char far *image);
```

_getlinestyle — Получает текущий стиль линии

```
#include <graph.h>
```

```
unsigned short far _getlinestyle(void);
```

_getphyscoord — Преобразует относительные координаты точки в абсолютные

```
#include <graph.h>
```

```
struct xycoord far _getphyscoord(short x, short y);
```

_getpixel, _getpixel_w — Получают значение пикселя

```
#include <graph.h>
```

```
short far _getpixel(short x, short y);
```

```
short far _getpixel_w(double wx, double wy);
```


_gettextcolor — Получает текущий цвет текста

```
#include <graph.h>
```

```
short far _gettextcolor(void);
```

_gettextcursor — Получает текущий атрибут курсора

```
#include <graph.h>
```

```
short far _gettextcursor(void);
```

_gettextposition — Получает текущую позицию текста

```
#include <graph.h>
```

```
struct rccoord far _gettextposition(void);
```

_getvideoconfig — Получает информацию о текущей графической среде

```
#include <graph.h>
```

```
struct videoconfig far * far _getvideoconfig(struct videoconfig far *config);
```

_getviewcoord, _getviewcoord_w, _getviewcoord_wxy — Преобразуют абсолютные или масштабированные координаты точки в относительные

```
#include <graph.h>
```

```
struct xycoord far _getviewcoord(short x, short y);
```

```
struct xycoord far _getviewcoord_w(double wx, double wy);
```

```
struct xycoord far _getviewcoord_wxy(struct _wxycoord * pwxyl);
```

_getvisualpage — Получает номер текущей видео страницы

```
#include <graph.h>
```

```
short far _getvisualpage(void);
```

_getwindowcoord — Преобразует относительные координаты точки в масштабированные

```
#include <graph.h>
```

```
struct _wxycoord far _getwindowcoord(short x, short y);
```

_imagesize, _imagesize_w, _imagesize_wxy — Получают размер буфера памяти для сохранения изображения

```
#include <graph.h>
```

```
long far _imagesize(short x1, short y1, short x2, y2);
```

```
long far _imagesize_w(double wx1, double wy1, double wx2, double wy2);
```

```
long far _imagesize_wxy (struct _wxycoord * pwxyl, struct _wxycoord * pwxyl2);
```

_lineto, _lineto_w — Вычерчивают прямую линию до определенной точки

```
#include <graph.h>
```

```
short far _lineto(short x, short y);
```

```
short far _lineto_w(double wx, double wy);
```

_moveto, _moveto_w — Перемещают текущую графическую позицию

```
#include <graph.h>
```

```
struct xycoord far _moveto(short x, short y);
```

```
struct _wxycoord far _moveto_w(double wx, double wy);
```

_outgtext — Печатает текст, использующий шрифты, в графическом режиме

```
#include <graph.h>
```

```
void far _outgtext(unsigned char far *text);
```


_outtext — Выводит цветной текст с текущей текстовой позиции

```
#include <graph.h>
```

```
void far _outtext(unsigned char far *text);
```

_pg_analyzechart, _pg_analyzechartms — Анализируют одиночные (многократные) ряды данных

```
#include <pgchart.h>
```

```
short far _pg_analyzechart(chartenv far * env,  
    char * far * categories, float far * values, short n);
```

```
short far _pg_analyzechartms(chartenv far * env, char * far * categories,  
    float * farvalues, short nseries,  
    short n, short arraydim, char * far * serieslabels);
```

_pg_analyzepie — Анализирует одиночные ряды данных для круговой диаграммы

```
#include <pgchart.h>
```

```
short far _pg_analyzepie(chartenv far * env,  
    char * far * categories, float far * values,  
    short far * explode, short n);
```

_pg_analyzescatter, _pg_analyzescatterms — Анализируют одиночные (многократные) ряды данных для точечной диаграммы

```
#include <pgchart.h>
```

```
short far _pg_analyzescatter(chartenv far * env,  
    float far * xvalues, float far * yvalues, short n);
```

```
short far _pg_analyzescatterms(chartenv far * env, float far * xvalues,  
    float far * yvalues, short nseries,  
    short n, short rowdim, char * far * serieslabels);
```

_pg_chart — Выдает однорядную диаграмму

```
#include <pgchart.h>
```

```
short far _pg_chart(chartenv far * env,  
    char * far * categories, float far * values, short n);
```

_pg_chartms — Выдает многорядную диаграмму

```
#include <pgchart.h>
```

```
short far _pg_chartms(chartenv far * env,  
    char * far * categories, float far * values, short nseries,  
    short n, short arraydim, char * far * serieslabels);
```

_pg_chartpie — Выдает круговую диаграмму

```
#include <pgchart.h>
```

```
short far _pg_chartpie(chartenv far * env,  
    char * far * categories, float far * values, short far * explode, short n);
```

_pg_chartscatter, _pg_chartscatterms — Выдают однорядную (многорядную) точечную диаграмму

```
#include <pgchart.h>
```

```
short far _pg_chartscatter(chartenv far * env,  
    float far * xvalues, float far * yvalues, short n);
```

```
short far _pg_chartscatterms(chartenv far * env,  
    float far * xvalues, float far * yvalues, short nseries,  
    short n, short rowdim, char * far * serieslabels);
```


_pg_defaultchart — Инициализирует диаграммную среду

```
#include <pgchart.h>
```

```
short far _pg_defaultchart(chartenv far * env,  
                           short charttype, short chartstyle);
```

_pg_getchardef — Получает пиксельную (битовую) схему для указанного символа

```
#include <pgchart.h>
```

```
void far _pg_getchardef(short charnum,  
                       charmap far * chardef);
```

_pg_getpalette — Получает цвета палитры, стили линий и шаблоны

```
#include <pgchart.h>
```

```
short far _pg_getpalette(paletteentry far * palette);
```

_pg_getstyleset — Получает текущее множество стилей

```
#include <pgchart.h>
```

```
void far _pg_getstyleset(short far * styleset);
```

_pg_hlabelchart — Записывает текст в окне горизонтально

```
#include <pgchart.h>
```

```
short far _pg_hlabelchart(chartenv far * env, short x,  
                          short y, short color, char far * label);
```

_pg_initchart — Инициализирует графические функции диаграмм

```
#include <pgchart.h>
```

```
short far _pg_initchart(void);
```

_pg_resetpalette — Восстанавливает цвета палитры, стили линий и шаблоны на значения по умолчанию

```
#include <pgchart.h>
```

```
short far _pg_resetpalette(void);
```

_pg_resetstyleset — Восстанавливает множество стилей на значения по умолчанию

```
#include <pgchart.h>
```

```
void far _pg_resetstyleset(void);
```

_pg_setchardef — Устанавливает пиксельную (битовую) схему для указанного символа

```
#include <pgchart.h>
```

```
void far _pg_setchardef(short charnum,  
                       charmap far * chardef);
```

_pg_setpalette — Устанавливает цвета палитры, стили линий и шаблоны

```
#include <pgchart.h>
```

```
short far _pg_setpalette(paletteentry far * palette);
```

_pg_setstyleset — Устанавливает текущее множество стилей

```
#include <pgchart.h>
```

```
void far _pg_setstyleset(short far * styleset);
```

_pg_vlabelchart — Записывает текст на экране вертикально

```
#include <pgchart.h>
```

```
short far _pg_vlabelchart(chartenv far * env, short x,  
                          short y, short color, char far * label);
```


_pie, _pie_wxy — Вычерчивают эллиптический клин ("кусочек пирога")

```
#include <graph.h>
```

```
short far _pie(short control, short x1, short y1, short x2,  
               short y2, short x3, short y3, short x4, short y4);
```

```
short far _pie_w(short control,  
                 double wx1, double wy1, double wx2, double wy2,  
                 double wx3, double wy3, double wx4, double wy4);
```

```
short far _pie_wxy(short control, struct _wxycoord * pwxyl,  
                   struct _wxycoord * pwxly2, struct _wxycoord * pwxly3,  
                   struct _wxycoord * pwxly4);
```

```
control:    _GFillInterior    _GBorder
```

_putimage, _putimage_w — Извлекают изображение из буфера

```
#include <graph.h>
```

```
void far _putimage(short x, short y, char far *image, short action);
```

```
void far _putimage_w(double wx, double wy, char far *image, short action);
```

```
action:    _GAND    _GOR    _GXOR    _GPSET    _GPRESET
```

_rectangle, _rectangle_w, _rectangle_wxy — Вычерчивают прямоугольник

```
#include <graph.h>
```

```
short far _rectangle(short control, short x1, short y1, short x2, short y2);
```

```
short far _rectangle_w(short control,  
                       double wx1, double wy1, double wx2, double wy2);
```

```
short far _rectangle_wxy(short control,  
                          struct _wxycoord * pwxyl, struct _wxycoord * pwxly2);
```

```
control:    _GFillInterior    _GBorder
```

_registerfonts — Инициализирует графическую систему шрифтов

```
#include <graph.h>
```

```
short far _registerfonts(unsigned char far * filename);
```

_remapallpalette — Переопределяет все цвета палитры

```
#include <graph.h>
```

```
short far _remapallpalette(long far * colors);
```

```
colors:    _BLACK    _BLUE    _GREEN    _CYAN    _RED    _MAGENTA    _BROWN  
           _WHITE    _GRAY    _LIGHTBLUE    _LIGHTGREEN    _LIGHTCYAN  
           _LIGHTRED    _LIGHTMAGENTA    _LIGHTYELLOW    _BRIGHTWHITE
```

_remappalette — Переопределяет отдельный цвет палитры

```
#include <graph.h>
```

```
long far _remappalette(short index, long color);
```

```
color:     _BLACK    _BLUE    _GREEN    _CYAN    _RED    _MAGENTA    _BROWN  
           _WHITE    _GRAY    _LIGHTBLUE    _LIGHTGREEN    _LIGHTCYAN  
           _LIGHTRED    _LIGHTMAGENTA    _LIGHTYELLOW    _BRIGHTWHITE
```

_selectpalette — Выбирает палитру для графических функций

```
#include <graph.h>
```

```
short far _selectpalette(short number);
```

_setactivepage — Устанавливает область памяти для записи изображений

```
#include <graph.h>
```

```
short far _setactivepage(short page);
```

_setbkcolor — Устанавливает текущий цвет фона

```
#include <graph.h>
```

```
long far _setbkcolor(long color);
```


_setcliprgn — Устанавливает область вывода для графических функций (порт экрана)

```
#include <graph.h>
```

```
void far _setcliprgn(short x1, short y1, short x2, short y2);
```

_setcolor — Устанавливает текущий цвет

```
#include <graph.h>
```

```
short far _setcolor(short color);
```

_setfillmask — Устанавливает маску закрашивания

```
#include <graph.h>
```

```
void far _setfillmask(unsigned char far * mask);
```

_setfont — Находит единственный шрифт с указанными характеристиками

```
#include <graph.h>
```

```
short far _setfont(unsigned char far * options);
```

_setlinestyle — Устанавливает маску стиля линии

```
#include <graph.h>
```

```
void far _setlinestyle(unsigned short mask);
```

_setpixel, _setpixel_w — Устанавливают значение пикселя

```
#include <graph.h>
```

```
short far _setpixel(short x, short y);
```

```
short far _setpixel_w(double wx, double wy);
```

_settextcolor — Устанавливает текущий цвет текста

```
#include <graph.h>
```

```
short far _settextcolor(short index);
```

_settextcursor — Устанавливает текущий атрибут курсора

```
#include <graph.h>
```

```
short far _settextcursor(short attr);
```

_settextposition — Устанавливает позицию текста

```
#include <graph.h>
```

```
struct rccoord far _settextposition(short row, short column);
```

_settextrows — Устанавливает число строк текста для текстовых режимов

```
#include <graph.h>
```

```
short far _settextrows(short rows);
```

_settextwindow — Создает текстовое окно

```
#include <graph.h>
```

```
void far _settextwindow(short r1, short c1, short r2, short c2);
```

_setvideomode — Устанавливает видеорежим

```
#include <graph.h>
```

```
short far _setvideomode(short mode);
```

mode: _DEFAULTMODE	_TEXTBW40	_TEXTC40	_TEXTBW80
_TEXTC80	_MRES4COLOR	_MRESNOCOLOR	_HRESBW
_TEXTMONO	_HERCMONO	_MRES16COLOR	_HRES16COLOR
_ERESNOCOLOR	_ERESCOLOR	_VRES2COLOR	_VRES16COLOR
_MRES256COLOR	_ORESCOLOR		

_setvideomoderows — Устанавливает видеорежим и число строк текста для текстовых режимов

```
#include <graph.h>
short far _setvideomoderows(short mode, short rows);
mode:  _DEFAULTMODE    _TEXTBW40    _TEXTC40    _TEXTBW80
        _TEXTC80        _MRES4COLOR  _MRESNOCOLOR _HRESBW
        _TEXTMONO       _HERCMONO    _MRES16COLOR _HRES16COLOR
        _ERESNOCOLOR    _ERESCOLOR   _VRES2COLOR  _VRES16COLOR
        _MRES256COLOR   _ORESCOLOR
```

_setvieworg — Помещает начало координат в точку с указанными абсолютными координатами

```
#include <graph.h>
struct xycoord far _setvieworg(short x, short y);
```

_setviewport — Устанавливает область вывода для графических функций (порт экрана) с относительной системой координат

```
#include <graph.h>
void far _setviewport(short x1, short y1, short x2, short y2);
```

_setvisualpage — Устанавливает область памяти для вывода изображений

```
#include <graph.h>
short far _setvisualpage(short page);
```

_setwindow — Устанавливает систему масштабированных координат

```
#include <graph.h>
short far _setwindow(short finvert, double wx1, double wy1,
                     double wx2, double wy2);
```

_unregisterfonts — Освобождает память, используемую шрифтами

```
#include <graph.h>
void far _unregisterfonts(void);
```

_wrapon — Разрешает или запрещает перенос строки

```
#include <graph.h>
short far _wrapon(short option);
option:  _GWRAPOFF  _GWRAPON
```

3. Функции работы с окнами

active_wn — Проверяет наличие окна на экране

```
#include <wios.h>
#include <window.h>
int active_wn(WINDOW *wn);
```

adj_cs — Проверяет позицию виртуального курсора

```
#include <wios.h>
#include <window.h>
int adj_cs(WINDOW *wn);
```

bell — Выдает звуковой сигнал

```
#include <wios.h>
#include <window.h>
void bell(void);
```


cls — Очищает экран

```
#include <wios.h>
#include <window.h>
void cls(void);
```

cl_wn — Очищает окно

```
#include <wios.h>
#include <window.h>
void cl_wn(WINDOW *wn);
```

color_sc — Устанавливает атрибут цвета для использования при очистке экрана

```
#include <wios.h>
#include <window.h>
void color_sc(char bground);
```

color_wn — Устанавливает цвет фона и текста в окне

```
#include <wios.h>
#include <window.h>
void color_wn(char fground, char bground, WINDOW *wn);
```

copy_wc — Копирует содержимое окна в файл

```
#include <wios.h>
#include <window.h>
int copy_wc(char dimen, char filename, char dmode, WINDOW *wn);
    dimen:  INSIDE  FULL
    mode:   FWRITE  FAPPEND
```

csr_hide, csr_show — Делают скрытым (видимым) курсор экрана

```
#include <wios.h>
#include <window.h>
void csr_hide(void);
void csr_show(void);
```

csr_type — Управляет размером и стилем курсора

```
#include <wios.h>
#include <window.h>
void csr_type(int type);
    type:  LINE  BLOCK  BOT_BLK  TOP_BLK
```

c_att — Устанавливает в байте атрибутов значение атрибута для цветов

```
#include <wios.h>
#include <window.h>
int c_att(char fground, char bground);
```

defs_wn, def_wn — Определяют окно

```
#include <wios.h>
#include <window.h>
void defs_wn(WINDOW *wn, int rb, int cb, int rw_q, int co_q, BORDER *bdr);
void def_wn(WINDOW *wn, int rb, int re, int cb, ce, int l_mg, int r_mg, BORDER *bdr);
    bdr:  BDR_OP  BDR_LNP  BDR_DLNP  BDR_REVP  BDR_DOTP  BDR_ASTP
```

def_fr — Инициализирует файл памяти

```
#include <wios.h>
#include <window.h>
int def_fr(FREC *mfile, char fname, int fmaxline, int fmaxcol);
```


dim_wn — Настраивает рабочую область окна

```
#include <wios.h>
#include <window.h>
int dim_wn(char dimen, WINDOW *wn);
    dimen: FULL INSIDE
```

di_file — Пересылает файл с диска в файл памяти

```
#include <wios.h>
#include <window.h>
int di_file(FREC *mfile);
```

di_st — Ввод с диска в строку

```
#include <stdio.h>
#include <wios.h>
#include <window.h>
int di_st(FILE fp, char stp, int q, char brk_ch, int tab_q);
```

dup_wn — Дублирует структуру окна

```
#include <wios.h>
#include <window.h>
void dup_wn(WINDOW *dwn, WINDOW *swn);
```

errout — Выдает сообщение об ошибке и обеспечивает выход из программы

```
#include <wios.h>
#include <window.h>
void errout(char * s1, char * s2);
```

file_lnp — Возвращает указатель на строку, соответствующую i-ой строке файла

```
#include <wios.h>
#include <window.h>
char *file_lnp(int frow, FREC *mfile);
```

free_file — Очищает содержимое файла памяти

```
#include <wios.h>
#include <window.h>
int free_file(FREC *fr);
```

free_mem — Освобождает память и устанавливает код ошибки

```
#include <wios.h>
#include <window.h>
int free_mem(char * p);
```

get_image — Возвращает указатель на видеостроку, содержащую образ окна

```
#include <wios.h>
#include <window.h>
char * get_image(WINDOW * wn);
```

get_mem — Получает память для сохранения образа окна

```
#include <wios.h>
#include <window.h>
char *get_mem(int size);
```

init_wns — Инициализирует систему работы с окнами

```
#include <wios.h>
#include <window.h>
void init_wns(void);
```


ki — Получает код клавиши от клавиатуры

```
#include <wios.h>
#include <window.h>
int ki(void);
```

ki_chk — Проверяет буфер клавиатуры на наличие кодов клавиш

```
#include <wios.h>
#include <window.h>
int ki_chk(void);
```

ki_cum — Получает одинаковые коды из буфера

```
#include <wios.h>
#include <window.h>
int ki_cum(KEYR * keyrec);
```

k_vcom — Транслирует коды специальных клавиш в команды управления прокруткой в памяти

```
#include <wios.h>
#include <window.h>
int k_vcom(int kv, int kq, WINDOW * wn);
```

pmenu — Выводит меню и обеспечивает выбор

```
#include <wios.h>
#include <window.h>
int pmenu(WINDOWPTR wnp, int qitems, int itemlength,
          int item_row, int def_pos);
```

mode_col — Получает число столбцов для текущего видеорежима

```
#include <wios.h>
#include <window.h>
int mode_col(void)
```

mod_wn — Изменяет координаты окна

```
#include <wios.h>
#include <window.h>
void mod_wn(int rb, int cb, int rowq, int colq, WINDOW * wn);
```

mv_cs — Изменяет положение виртуального курсора

```
#include <wios.h>
#include <window.h>
void mv_cs(int rw, int co, WINDOW * wn);
```

mv_csr — Изменяет положение виртуального курсора и перемещает курсор экрана в позицию виртуального курсора

```
#include <wios.h>
#include <window.h>
void mv_csr(int rw, int co, WINDOW * wn);
dir: UP .DOWN
```

mv_rws — Прокручивает текст в окне в заданном направлении

```
#include <wios.h>
#include <window.h>
int mv_rws(int nlines, char dir, WINDOW * wn);
```

mv_wi — Перемещает окно в новую позицию на экране

```
#include <wios.h>
#include <window.h>
int mv_wi(int rw, int co, WINDOW * wn);
```


pl_csr — Устанавливает курсор экрана в позицию виртуального курсора

```
#include <wios.h>
#include <window.h>
void pl_csr(WINDOW * wn);
```

pl_mfwn — Перемещает окно в заданную позицию

```
#include <wios.h>
#include <window.h>
int pl_mfwn(int frow, int fcol, WINDOW * wn);
```

pl_wn — Устанавливает новые значения элементов, определяющих положение окна

```
#include <wios.h>
#include <window.h>
void pl_wn(int rw, int co, WINDOW * wnp);
```

prt_wn — Копирует содержимое окна на печать

```
#include <wios.h>
#include <window.h>
int prt_wn(WINDOW * wn);
```

rd_csr — Читает координаты курсора

```
#include <wios.h>
#include <window.h>
void rd_csr(int * row, int * col, int page);
```

rd_mode — Возвращает текущий режим видеомонитора

```
#include <wios.h>
#include <window.h>
int rd_mode(void);
```

repl_wi — Помещает сохраненный образ окна на экран

```
#include <wios.h>
#include <window.h>
int repl_wi(WINDOW * wn);
```

sav_image — Связывает видеостроку, содержащую образ окна, со структурой, определяющей окно

```
#include <wios.h>
#include <window.h>
void sav_image(char * vstr, WINDOW * wn);
```

sav_wi — Сохраняет образ окна в памяти

```
#include <wios.h>
#include <window.h>
int sav_wi(WINDOW * wn);
```

scrl_file — Прокручивает файл памяти

```
#include <wios.h>
#include <window.h>
int scrl_file(int nrow, int dir, int beg_row, FREQ * mfile);
dir: UP .DOWN
```

set_wn — Устанавливает окно на экране

```
#include <wios.h>
#include <window.h>
int set_wn(WINDOW * wn);
```


size_wn — Вычисляет размер видеостроки для сохранения образа окна

```
#include <wios.h>
#include <window.h>
int size_wn(char dimen, int part, WINDOW * wn);
    dimen:    FULL    INSIDE
    path:     CH      ENDROW ROW COL  ENDCOL WIND
```

sti_file — Помещает строку в файл памяти

```
#include <wios.h>
#include <window.h>
int sti_file(char * st, int frow, FREC * mfile);
```

lower_st — Преобразует каждую позицию строки в строчную букву

```
#include <wios.h>
#include <window.h>
void lower_st(char * st);
```

skip_wh — Пропускает ведущие пробелы в строке и возвращает указатель на первый непробельный символ;

```
#include <wios.h>
#include <window.h>
char * skip_wh(char * st);
```

stblank — Распределяет память для размещения строки и завершающего символа строки ('\0') и инициализирует строку пробелами

```
#include <wios.h>
#include <window.h>
char * stblank(int len);
```

strcpyp — Копирует содержимое исходной строки в строку-адресат

```
#include <wios.h>
#include <window.h>
char * strcpyp(char * dest, char * src);
```

strip_wh — Удаляет конечные пробелы в строке и перемещает завершающий нулевой символ ('\0')

```
#include <wios.h>
#include <window.h>
void strip_wh(char * st);
```

upper_st — Преобразует каждую позицию строки в прописную букву

```
#include <wios.h>
#include <window.h>
void upper_st(char * st);
```

sw_att — Устанавливает указанный цвет для окна

```
#include <wios.h>
#include <window.h>
void sw_att(int att, WINDOW * wn);
```

sw_bdratt — Устанавливает указанный цвет для рамки окна

```
#include <wios.h>
#include <window.h>
void sw_bdratt(int att, WINDOW * wn);
```


sw_border — Устанавливает в элементе `wn.bdrp` указатель на заданный тип рамки

```
#include <wios.h>
```

```
#include <window.h>
```

```
void sw_border (BORDER * bdr, WINDOW * wn);
```

sw_clear — Устанавливает переключатель очистки конца строки в состояние ON или OFF

```
#include <wios.h>
```

```
#include <window.h>
```

```
void sw_clear(int state, WINDOW *wn);
```

sw_csadv — Устанавливает переключатель продвижения виртуального курсора в состояние ON или OFF

```
#include <wios.h>
```

```
#include <window.h>
```

```
void sw_csadv(int state, WINDOW * wn);
```

sw_latt — Устанавливает в элементе `wn.larray` указатель на заданный массив логических атрибутов

```
#include <wios.h>
```

```
#include <window.h>
```

```
void sw_latt(char att_array, WINDOW * wn);
```

sw_margin — Устанавливает левый и правый отступ от рамки

```
#include <wios.h>
```

```
#include <window.h>
```

```
void sw_margin(int l_mg, int r_mg, WINDOW * wn);
```

sw_mfile — Устанавливает в элементе `wn.fgr` указатель на заданную структуру файла памяти

```
#include <wios.h>
```

```
#include <window.h>
```

```
void sw_mfile(FREC * mfile, WINDOW * wn);
```

sw_name — Устанавливает имя окна

```
#include <wios.h>
```

```
#include <window.h>
```

```
void sw_name(char * name, WINDOW * wn);
```

sw_plcsr — Устанавливает переключатель перемещения курсора экрана в состояние ON или OFF

```
#include <wios.h>
```

```
#include <window.h>
```

```
void sw_plcsr(int state, WINDOW * wn);
```

sw_wntyp — Устанавливает тип окна — стековое или простое

```
#include <wios.h>
```

```
#include <window.h>
```

```
void sw_wntyp(int state, WINDOW * wn);
```

state: POPUP ORDINARY

sw_scroll — Включает или отключает автоматическую прокрутку окна

```
#include <wios.h>
```

```
#include <window.h>
```

```
void sw_scroll(int state, WINDOW * wn);
```


sw_wwrap — Устанавливает переключатель переноса целого слова в состояние ON или OFF

```
#include <wios.h>
#include <window.h>
void sw_wwrap(int state, WINDOW * wn);
```

s_latt — Заполняет массив логических атрибутов

```
#include <wios.h>
#include <window.h>
int s_latt(int col, int rowq, int colq, char * att_tbl, char * att_arr);
```

unsav_wi — Помещает образ окна на экран

```
#include <wios.h>
#include <window.h>
int unsav_wi(WINDOW * wn);
```

unset_wn — Удаляет окно с экрана

```
#include <wios.h>
#include <window.h>
int unset_wn(WINDOW * wn)
```

u_init — Инициализирует глобальные переменные

```
#include <wios.h>
#include <window.h>
void u_init(void);
```

vid_bdr — Устанавливает цвет рамки экрана

```
#include <wios.h>
#include <window.h>
void vid_bdr(int color);
```

vid_int — Выполняет прерывание BIOS 10H (ввод-вывод на экран дисплея)

```
#include <wios.h>
#include <window.h>
void vid_int(VIDIO * vri, VIDIO * vro);
```

vid_mode — Устанавливает режим экрана и обеспечивает переключение адаптеров дисплеев

```
#include <wios.h>
#include <window.h>
void vid_mode(int mode);
```

vo_att — Читает атрибут окна

```
#include <wios.h>
#include <window.h>
int vo_att(WINDOW * wn);
```

vo_ch — Читает символ из окна

```
#include <wios.h>
#include <window.h>
int vo_ch(WINDOW * wn);
```

vs_file — Отображает в окне и прокручивает файл

```
#include <wios.h>
#include <window.h>
void vs_file(int exit_key, WINDOW * wn);
```


v_att — Записывает указанный видеоатрибут в текущую позицию виртуального курсора

```
#include <wios.h>
#include <window.h>
int v_att(int att, WINDOW * wn);
```

v_axes — Вычерчивает оси для диаграмм

```
#include <wios.h>
#include <window.h>
int v_axes(int r_origin, int c_origin, int height, int width,
           WINDOW * wn);
```

v_bar — Вычерчивает горизонтальную или вертикальную полосу

```
#include <wios.h>
#include <window.h>
int v_bar(int row_size, int col_size, int r_begin,
          int c_begin, char ch, char attrib, WINDOW * wn,
          BORDER * bdr);
```

v_border — Вычерчивает рамку окна

```
#include <wios.h>
#include <window.h>
void v_border(WINDOW * wn, BORDER * bdr);
```

v_ch — Выводит символ в окно

```
#include <wios.h>
#include <window.h>
int v_ch(int ch, WINDOW * wn);
```

v_co — Выводит столбец символов в окно

```
#include <wios.h>
#include <window.h>
int v_co(char ch, int q, WINDOW * wn);
```

v_file — Отображает файл памяти в окне

```
#include <wios.h>
#include <window.h>
void v_file(WINDOW * wn);
```

v_fst — Помещает полную строку в окно

```
#include <wios.h>
#include <window.h>
void v_fst(char * st, WINDOW * wn);
```

v_mov — Пересылает информацию между видеобуфером и видеострокой

```
#include <wios.h>
#include <window.h>
int v_mov(char * vst, WINDOW * wn, int part, int direct);
part:     CH    ENDROW ROW  COL  ENDCOL WIND  ENDWIND
direct:   IN    OUT
```

v_mova — Пересылает символы между окном и стандартной строкой Си

```
#include <wios.h>
#include <window.h>
int v_mova(char * st, WINDOW * wn, int part, int direct);
part:     CH    ENDROW ROW  COL  ENDCOL WIND  ENDWIND
direct:   IN    OUT
```


v_natt — Устанавливает атрибуты для части окна

```
#include <wios.h>
#include <window.h>
int v_natt(char att, int part, WINDOW * wn);
part:      CH   ENDROW ROW  COL   ENDCOL WIND  ENDWIND
```

v_plst — Пересылает строку в окно

```
#include <wios.h>
#include <window.h>
char *v_plst(int row, int col, char * st, WINDOW * wn);
```

v_printf — Выполняет форматизованный вывод в окно

```
#include <wios.h>
#include <window.h>
void v_printf(WINDOW * wn, char * fmt, ...);
```

v_qch — Выводит пары "символ-атрибут" в окно

```
#include <wios.h>
#include <window.h>
int v_qch(char ch, int q, WINDOW * wn);
```

v_rw — Выводит в окно строку символов

```
#include <wios.h>
#include <window.h>
int v_rw(char ch, int q, WINDOW * wn);
```

v_st — Выводит в окно строку символов с прокруткой

```
#include <wios.h>
#include <window.h>
char *v_st(char * st, WINDOW * wn);
```

v_st_nop — Выводит заданное количество символов строки в окно без учета специальных символов

```
#include <wios.h>
#include <window.h>
void v_st_nop(char * st, int q, WINDOW * wn);
```

v_st_rw — Выводит строку в строку окна с учетом специальных символов

```
#include <wios.h>
#include <window.h>
char *v_st_rw(char * st, int q, WINDOW * wn);
```


ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ В СИСТЕМУ ПРОГРАММИРОВАНИЯ СИ	3
Ч А С Т Ь 1. ЯЗЫК ПРОГРАММИРОВАНИЯ СИ	11
Г л а в а 1. ЭЛЕМЕНТЫ ЯЗЫКА	11
1.1. Классификация символов языка	11
1.2. Лексемы	14
Г л а в а 2. ТИПЫ ДАННЫХ. ОБЪЯВЛЕНИЯ	20
2.1. Категории типов данных	20
2.2. Целые типы данных	21
2.3. Плавающие типы данных	24
2.4. Указатели	25
2.5. Перечисления	26
2.6. Массивы	29
2.7. Структуры	31
2.8. Смеси	34
2.9. Объявления объектов и типов	36
2.10. Инициализация данных	39
Г л а в а 3. ВЫРАЖЕНИЯ И ПРИСВАИВАНИЯ	42
3.1. Операнды	43
3.2. Операции	49
3.3. Приоритеты операций и порядок вычисления	63
3.4. Побочные эффекты	65
3.5. Преобразование типов	66
Г л а в а 4. ОПЕРАТОРЫ	69
4.1. Оператор "выражение"	69
4.2. Пустой оператор	69
4.3. Составной оператор	70
4.4. Оператор if	70
4.5. Оператор switch	72
4.6. Оператор break	74
4.7. Оператор for	74
4.8. Оператор while	76
4.9. Оператор do	77
4.10. Оператор continue	78
4.11. Оператор return	78
4.12. Оператор goto	79

Глава 5.	ФУНКЦИИ	80
5.1.	Основные понятия	80
5.2.	Определение функции	82
5.3.	Объявление функции (прототип)	86
5.4.	Вызовы функций	87
5.5.	Стандартные функции библиотеки Си	95
Глава 6.	СТРУКТУРА ПРОГРАММЫ И КЛАССЫ ПАМЯТИ	98
6.1.	Основные определения	98
6.2.	Объявление переменных на внутреннем уровне	100
6.3.	Объявление переменных на внешнем уровне	103
6.4.	Объявление функции	104
6.5.	Правила определения времени жизни и области видимости объекта	105
6.6.	Правила инициализации для внешних и внутренних переменных	106
6.7.	Классы имен	107
Глава 7.	ДИРЕКТИВЫ ПРЕПРОЦЕССОРА И ПРАГМЫ	108
7.1.	Директива <code>#define</code>	109
7.2.	Препроцессорные операции	112
7.3.	Директива <code>#undef</code>	114
7.4.	Директива <code>#include</code>	114
7.5.	Директивы условной компиляции	116
7.6.	Директивы <code>#line</code> и <code>#error</code>	117
7.7.	Прагмы	119
Часть 2.	НЕКОТОРЫЕ РЕКОМЕНДАЦИИ ПО ПРОГРАММИРОВАНИЮ	121
Глава 8.	УКАЗАТЕЛИ И АДРЕСНАЯ АРИФМЕТИКА	121
8.1.	Определение указателей	121
8.2.	Указатель на неопределенный тип	123
8.3.	Выражения с указателями	124
8.4.	Преобразование указателей	127
8.5.	Связь указателей и массивов	127
Глава 9.	МОДЕЛИ ПАМЯТИ	132
9.1.	Некоторые сведения о процессоре	132
9.2.	Организация памяти и адресация	133
9.3.	Структура объектной Си-программы	134
9.4.	Адресация <code>near</code> , <code>far</code> и <code>huge</code>	135
9.5.	Стандартные модели памяти	138
9.6.	Размер указателей в стандартных моделях памяти	143
9.7.	Изменение размера указателей для стандартных моделей памяти	144
9.8.	Особенности работы с большими массивами	148
9.9.	Преобразование указателей при вызове функций	149

Глава 10. ОРГАНИЗАЦИЯ СВЯЗИ С ПРОГРАММАМИ НА ДРУГИХ ЯЗЫКАХ	151
10.1. Соглашения о связях и именах в Си	153
10.2. Соглашения о связях и именах в Паскале и Фортране	153
10.3. Организация связи с программами на языке ассемблера	154
10.4. Написание ассемблерной процедуры	155
10.5. Организация связи с программами на Паскале и Фортране	171
ЧАСТЬ 3. ИСПОЛЬЗОВАНИЕ ФУНКЦИЙ БИБЛИОТЕКИ ПРИ ПРОГРАММИРОВАНИИ ПРИМЕНЕНИЙ	187
Глава 11. ВВОД И ВЫВОД	187
11.1. Ввод-вывод потока	187
11.2. Управление буферизацией	197
11.3. Двоичный и текстовый режимы	199
11.4. Форматизованный ввод и вывод	201
11.5. Ввод-вывод нижнего уровня	210
11.6. Ввод и вывод для консоли и порта	213
11.7. Другие функции ввода-вывода	217
11.8. Ограничения по использованию функций ввода-вывода	220
Глава 12. РАБОТА С ОКНАМИ	221
12.1. Определение окна	221
12.2. Операции с окнами	229
12.3. Просмотр файла в окне	236
12.4. Создание оконных меню	239
12.5. Создание справочной системы	241
Глава 13. ГРАФИКА	247
13.1. Структура графической программы	247
13.2. Доступ к конфигурации видеорежима	249
13.3. Система координат	251
13.4. Управление цветом	254
13.5. Нанесение точек и вычерчивание линий	263
13.6. Вычерчивание фигур	269
13.7. Перемещение фигур по экрану	273
ЧАСТЬ 4. ИНТЕГРИРОВАННАЯ СРЕДА РАЗРАБОТКИ СИ-ПРОГРАММ	277
Глава 14. ОСНОВНЫЕ ПОНЯТИЯ СРЕДЫ	277
14.1. Экран среды Си	277
14.2. Выбор команд меню	279
14.3. Кадры диалога	282
Глава 15. СОЗДАНИЕ И РЕДАКТИРОВАНИЕ ИСХОДНЫХ ПРОГРАММ	284
15.1. Операции встроенного текстового редактора	284
15.2. Меню "Текст"	287
15.3. Меню "Где"	288
15.4. Меню "Файл"	290

Глава 16. КОМПИЛЯЦИЯ, ВЫПОЛНЕНИЕ И ОТЛАДКА ПРОГРАММ	292
16.1. Меню "Режимы"	292
16.2. Меню "Проект"	294
16.3. Меню "Запуск"	296
16.4. Меню "Отладка"	298
16.5. Меню "Экран"	302
16.6. Меню "Утилиты"	304
16.7. Справочная система среды Си	306
Литература	307
Приложение 1. КЛАВИАТУРА ПЭВМ ЕС	308
Приложение 2. ПРОТОТИПЫ ФУНКЦИЙ БИБЛИОТЕКИ СП СИ	310

ВНИМАНИЮ ЧИТАТЕЛЕЙ!

Издательство "Финансы и статистика" совместно с предприятием "ЮНИТИ" выпустило в свет новую книгу:

Дей А.К. Ventura Publisher 2.0: Практическое пособие: Пер. с англ. – М.: Финансы и статистика, ЮНИТИ, 1991 (II кв.). – 12 л.: ил. – ISBN 5-279-00945-8 (в пер.): 10 р., 50 000 экз.

Прекрасно структурированное практическое пособие для пользователей одной из самых популярных в своем классе программ верстки для ПК. Книга охватывает базовый комплект пакетов Ventura Publisher и профессиональное расширение. Даются четкие определения базовых понятий программ. Описываются возможности подготовки полос издания, включающих многоколоночное размещение текста, рисунки, формулы и таблицы, создание и вывод на печать репродуцируемого оригинала-макета.

Для редакционно-издательских работников, а также практиков, занятых внедрением прогрессивных методов подготовки изданий различных типов.

Заявки на книгу направляйте по адресу: 103062, г. Москва, а/я 86.

Научное издание

**Романовская Лорина Михайловна
Русс Татьяна Васильевна
Свитковский Сергей Геннадиевич**

**Программирование в среде СИ
для ПЭВМ ЕС**

Зав. редакцией **И. Г. Дмитриева**
Редакторы **Л. Д. Григорьева, Л. А. Табакова**
Худож. редактор **С. Л. Витте**
Техн. редактор **Л. Г. Чельшева**
Корректор **Н. П. Сперанская**
Переплет художника **Е. К. Самойлова**
ИБ № 2678

Оригинал-макет подготовлен в СНПО "Алгоритм"

Сдано в набор 04.03.91. Подписано в печать 23.05.91. Формат 60 x 88 1/16.
Бум. офсетная. Гарнитура Таймс. Печать офсетная. Усл. п. л. 21,56.
Усл. кр.-отт. 21,56. Уч.-изд. л. 23,42. Доп. тираж 50 000 экз. Заказ 883.
Цена договорная.

Издательство "Финансы и статистика",
101000 Москва, ул. Чернышевского, 7.

Отпечатано с оригинал-макета в типографии им. Котлякова
издательства «Финансы и статистика» Государственного
комитета СССР по печати.
195273, С.-Петербург, ул. Руставели, 13.

